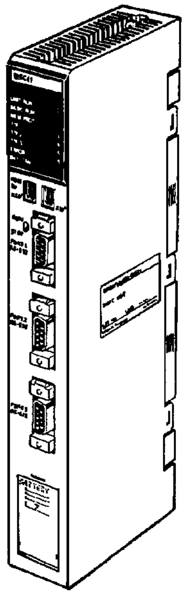
BASIC Units
CV500-BSC11/21/31/41/51/61

Reference Manual

Revised March 1993



CV500-BSC11

Notice:

OMRON products are manufactured for use according to proper procedures by a qualified operator and only for the purposes described in this manual.

The following conventions are used to indicate and classify warnings in this manual. Always heed the information provided with them.

Caution Indicates information that, if not heeded, could result in minor injury or damage to the product.

DANGER! Indicates information that, if not heeded, could result in loss of life or serious injury.

OMRON Product References

All OMRON products are capitalized in this manual. The word “Unit” is also capitalized when it refers to an OMRON product, regardless of whether or not it appears in the proper name of the product.

The abbreviation “Ch,” which appears in some displays and on some OMRON products, often means “word” and is abbreviated “Wd” in documentation in this sense.

The abbreviation “PC” means Programmable Controller and is not used as an abbreviation for anything else.

Visual Aids

The following headings appear in the left column of the manual to help you locate different types of information.

Note Indicates information of particular interest for efficient and convenient operation of the product.

1, 2, 3... 1. Indicates lists of one sort or another, such as procedures, checklists, etc.

© OMRON, 1992

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, mechanical, electronic, photocopying, recording, or otherwise, without the prior written permission of OMRON.

No patent liability is assumed with respect to the use of the information contained herein. Moreover, because OMRON is constantly striving to improve its high-quality products, the information contained in this manual is subject to change without notice. Every precaution has been taken in the preparation of this manual. Nevertheless, OMRON assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained in this publication.

TABLE OF CONTENTS

SECTION 1

SECTION 2

SECTION 3

About this Manual:

This manual describes the BASIC Unit instruction set and includes the sections described below.

Please read this manual completely and be sure you understand the information provided before attempting to operate the BASIC Unit.

Section 1 Describes the syntax and conventions used in the manual, and gives background information on the BASIC Unit.

Section 2 Contains a table listing all of the BASIC Unit instructions, with usage and a brief description.

Section 3 Lists each instruction in alphabetical order, with in-depth descriptions, examples, and sample programs.

SECTION 1
BASIC Syntax

1-1 Conventions

In this manual, sample command lines and program code fragments will be printed in this `typewriter font`. Words, numbers, and symbols in this font should be entered exactly as they are shown. *Italics* are used for filenames, variables, and placeholders (in which case, the programmer must select an element that matches the placeholder type). The _ character indicates a space.

1-2 Character Set

The following characters can be used with the BASIC Unit:

Classification	Characters
Alphabetic	`A` through `Z` (upper case), `a` through `z` (lower case)
Numeric	`0` through `9`
Special	Space (shown in the text as _) `! ″ # $ % & ′ ( ) * + − / , . : ; < > = ? [ ] \ ^ @ ~ _`

1-3 Keywords

A keyword is a word that has special meaning for the BASIC unit. Keywords direct the Unit to perform some action that is useful for the programmer. Keywords will always be shown in the `typewriter font`, using capital letters. In a program, keywords can appear in upper, lower, or mixed case. Examples of keywords are: `PRINT`, `GOTO`, and `END`.

Keywords are also called reserved words. The reserved words of the BASIC Unit are listed in *Appendix B Reserved Word List*.

1-4 Commands

Commands and statements (below) are both instructions that the BASIC Unit can execute. The difference is that commands are usually used in direct mode, while statements are usually used in programs.

1-5 Statements

A statement, such as `LET E=M*C^2`, is a sequence of BASIC keywords, variable names, constants, and symbols, that tells the Unit to perform a calculation or other action.

1-6 Lines

A program is a collection of lines, and each line consists of a line number and one or more statements:

```
10 PRINT ″BASIC UNIT″ ......... Executable statement

20 REM ***BASIC UNIT***   ..... Comment

30 ′ ***BASIC UNIT*** ......... Comment

40 PRINT : PRINT ″END″ ........ Two executable statements

50 END ........................ Executable statement
```

Statement

Line number

Lines must not exceed 255 characters in length. Several statements can be placed on a single line, provided that the total number of characters (including spaces) does not exceed 255. Each statement must be separated from the others by a colon (:).

```
10 INPUT A$ : PRINT A$ ..................... Correct

20 PRINT ”BASIC UNIT” ...................... Correct

30 PRINT A$ : ........... : PRINT Y$ ........ Correct
             255 characters or less
40 PRINT A$: ........... : PRINT Z$ ........ Incorrect
             More than 255 characters
```

A line with no statements or labels (a line number by itself) is automatically deleted.

1-6-1 Line Numbers

Line numbers must be unique integers between 1 and 65529. The program is normally executed in ascending order of the line numbers.

Line numbers are used in statements such as GOTO or GOSUB, which alter the normal flow of program execution.

1-6-2 Labels

Labels may be used instead of line numbers in GOTO and GOSUB statements. A label must appear as the first statement on a line, and it must start with an asterisk (*) and an alphabetic character (A to Z). The rest of the characters in the label can be letters, numbers, or periods. Here is an example of a label:

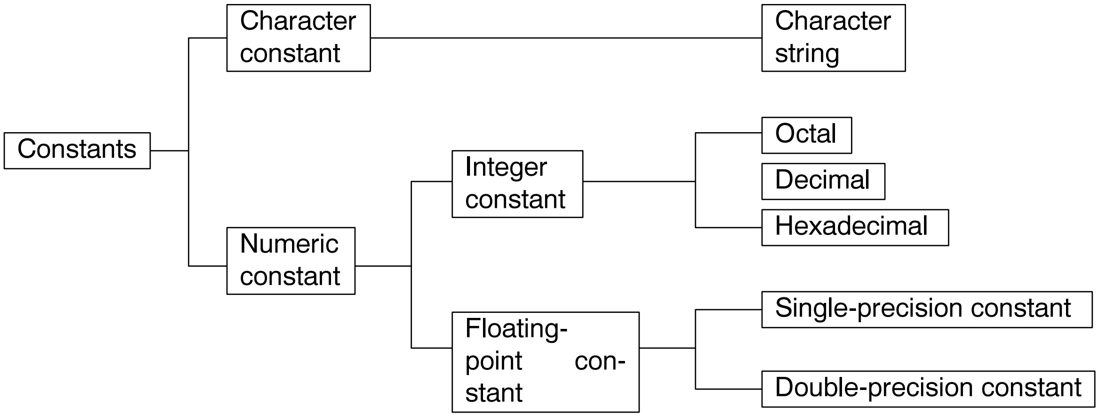
```
10 *This.is.a.label
```

Only the first 41 characters (including the *) of a label are significant. Characters after the 41st are ignored. Uppercase and lowercase letters are not distinguished, so the labels *ABC and *abc are considered identical.

Reserved words (keywords) may not be used as labels.

1-7 Constants

Constants are numbers or character strings which are directly coded in a program. There are various types of constants, as shown in this chart:



1-7-1 Character Constants

A character constant (character string) consists of alphanumeric characters, symbols, and spaces enclosed in a pair of double quotation marks (””). The length of the constant (i.e., the number of characters) is limited to the number of characters that can be written on one line (255).

To include a double quotation mark as a character in a character constant, the `CHR$` function must be used. A character string whose length is zero is called a null string. Here are some examples of character strings:

`"FACTORY_AUTOMATION"` .... This character constant consists of 17 alphabetic characters and 1 space.

`"1234567890"` .... Arithmetic operations on character strings of digits is not allowed.

`CHR$(34)` .... This is a character constant consisting of only one character, a double quotation mark.

`""` .... This is a null string.

1-7-2 Numeric Constants

A numeric constant is a number on which arithmetic operations can be performed. A plus (+) or minus (–) sign may be placed before a numeric constant to indicate whether the constant is positive or negative. (If the sign is omitted, the number is assumed to be positive.) Numeric constants can be broadly classified into integer constants and real-number constants.

Integer Constants

The BASIC Unit can handle integer constants written in octal, decimal, or hexadecimal notation.

Decimal

A decimal integer constant is written as a plain decimal number followed by a percent sign (`%`). The number must be in the range –32767 to 32767. If a percent sign is appended to a number with a fractional part, the number will be rounded to make an integer constant. Examples of decimal integer constants are shown below.

`21474%`

`–256%`

`123.456%` .... (This will be rounded to make 123%)

Note The BASIC Units treats all numeric constants as absolute numeric values. Therefore, although –32768 can be represented as a 16-bit value, the number cannot be directly entered as a decimal integer constant.

Octal

An octal constant starts with `&O` or `&` and consists of octal numbers (`0` to `7`). The constant must be no longer than 6 digits, excluding the `&O` or `&`, and must be between `&O0` and `&O177777`. Integer constants entered in the program as octal numbers will be converted to decimal if they are output to a display or printer. Examples of octal constants are:

`&O123` .... (decimal 83)

`&O1234` .... (decimal 668)

`&12345` .... (decimal 5349)

Hexadecimal

A hexadecimal constant starts with `&H`, followed by one to four hexadecimal digits (`0` to `9` and `A` to `F`). It must be in the range of `&H0` to `&HFFFF`. An integer constant entered in the program as a hexadecimal number will be converted into a decimal number if it is output to a display or printer. Here are some examples of hexadecimal constants:

`&H12` .... (decimal 18)

`&H2E3F` .... (decimal 11839)

Floating-point Constants

The real-number constants the BASIC Unit can handle are divided into two types: single-precision and double-precision.

Single-precision Constants

A single-precision constant is written as a decimal number with an optional exponent indicated by `e` or `E`. It is stored in a form that is accurate to about seven significant digits. When displayed, however, only six digits are output (the seventh digit is used to round properly). The allowable range for single-precision constants is approximately +3.4*10^{38} for substitution and +3.40282*10^{38} as the result of each operation. In addition, single-precision constants must satisfy one of the following conditions:

- 7 or fewer significant digits
- Exponent indicated with `e` or `E`
- Exclamation point (`!`) placed after constant

Examples of single-precision constants:

```
−32.11
6.471E−4 .......... (6.471*10−4 in standard notation)
+123.456789! ..... (more than 7 significant digits, but ! is specified)
```

Double-precision Constants

A double-precision constant is written the same as a single-precision constant, but the exponent (if any) is indicated with `d` or `D`. Double-precision constants are stored in a form that is accurate to about seventeen significant digits. When displayed, however, only sixteen digits are shown (the seventeenth digit is used to round properly). The allowable range for double-precision constants is about +1.07*10^{-308} to +1.07*10^{307}. Constants are made double-precision if they satisfy one of these conditions:

- More than 7 significant digits
- Exponent indicated with `d` or `D`
- Sharp mark (#) placed after constant

Examples of double-precision constants:

```
0.00000045
−3.1415D+13 ...... (–3.1415*1013 in standard notation)
123.45#
```

Caution The BASIC Unit normally treats all numeric constants that have no type specifier (`%`,`!`, or #, or `d` or `D` exponent) as single-precision constants. This may cause errors if the constants are outside the range for single-precision values.

1-8 Variables

A variable is a name you can use to represent a value in a BASIC program. The value of a variable can be set or changed with the `LET` statement. Before a variable has been set by the program, it has a value of 0 if it is a numeric variable, or a null string if it is a character variable.

Variables can be classified by type, data structure, purpose, and storage area as follows:

Type	Character variable (stores a character string)	Fixed-length character string (default: length is 18 characters)
		Variable-length character string (538 characters max.)
	Numeric variable (stores a numeric value)	Integer variable (2 bytes)
		Single-precision floating-point variable (4 bytes)
		Double-precision floating-point variable (8 bytes)
Data structure	Simple variable (variable with only one data value)	
	Array variable (variable with several data values in contiguous memory areas)	
Purpose	Local variable (variable that can be accessed in only one task)	
	Global variable (variable that can be accessed between tasks)	
Storage area	Volatile variable (ordinary variable that loses its contents when power is turned off)	
	Non-volatile variable (variable whose data is retained even after power has been turned off and is not initialized on initialization)	

For details on the structure of a variable, refer to *Appendix A Memory Storage Format of Variables*.

1-8-1 Variable Names and Types

A variable name can be up to 40 characters long. It must start with a letter (`A` to `Z`), and can be followed by letters, numbers, or periods. Uppercase and lowercase letters are not distinguished, so a variable called `VAR1` is considered to be the same as one called `var1`. Reserved words (keywords) cannot be used for variable names.

An name with no type specifier normally indicates a variable that can hold a single-precision floating-point value; if you want to store a character string, integer, or double-precision value, you must put a type specifier after the name. The characters `$`, `%`, and # indicate character strings, integers, and double-precision variables, respectively. An exclamation point (`!`) after a variable name indicates a single-precision floating-point value.

Here are some valid variable names and their types:

`HEIGHT ...........`	single-precision floating-point
`WEIGHT! ..........`	single-precision floating-point
`YOUR.NAME$ ........`	character string
`REJECT.PARTS% ....`	integer
`TINY.NUMBER# .....`	double-precision floating-point

Two variables of different types may use the same name: `ABC%` is a different variable than `ABC#`. However, a local variable may not use the same name as a global variable.

A variable name starting with `FN` is treated as the name of a user-defined function (refer to the description of the `DEF FN` statement).

Type Declaration

A variable name with no type specifier ordinarily indicates a single-precision floating-point value. The `DEFINT`, `DEFSNG`, `DEFDBL`, and `DEFSTR` statements can be used to change the default type for a single variable or for sets of variables whose names start with certain letters.

1-8-2 Simple Variables and Array Variables

A variable name used by itself (with or without a type specifier character) is called a simple variable. It is often convenient, however, to have a table of values associated with one name. Each element of the table, or *array*, can be referenced using the array name indexed with an integer or integer expression. The index is enclosed in parentheses. An array variable has as many indices as the array has dimensions.

Before using an array variable, you must declare the number of elements it will have with the `DIM` or `RDIM` statement.

Here are some examples of array declarations:

`10 DIM DAYS.PER.MONTH%(12)` An array of 12 integers

`20 DIM TIC.TAC.TOE(3,3)` ... An array of 9 single-precision floating-point values, arranged in three rows of three.

Each array index must be an integer from 0 to 32767. Normally, the first element of an array is element 0; this can be changed using the `OPTION BASE` statement.

The size of the array is equal to the size of one element multiplied by the number of elements. (See *Appendix A: Storage Format of Variables* for information about variable sizes.) You cannot declare an array that is larger than available memory.

If a variable is used with only a variable name specified, it serves only as a simple variable. If an index enclosed in ( ) is suffixed to the variable, however, the variable serves as an array variable.

An array variable must be explicitly declared by the `DIM` or `RDIM` instructions. `DIM` and `RDIM` can also be used to declare global and local variables, and to set the length of a fixed-length character string variable.

1-8-3 Character String Variables

Character string variables can normally contain strings of any length from 0 to 538 characters. The `DIM` statement can be used to declare fixed-length character strings. For example:

`DIM B$ 10` .... `B$` is a fixed-length character string variable which can hold no more than 10 characters.

If a character variable is declared with `DIM`, but no length is specified, the length is fixed at a system default value. This default value is normally 18 characters, but it can be changed with the `OPTION LENGTH` instruction.

Fixed-length strings may use memory less efficiently than variable-length strings, but the system has to do less processing to maintain a fixed-length string, so your program may run faster.

Examples using the `DIM` statement:

`DIM A` .... Simple variable of single-precision, real-number type

`DIM B$` .... Fixed-length character string variable (can hold 18 characters, or whatever default value has been set with the `OPTION LENGTH` instruction).

`DIM C$ 40` .... Fixed-length character string variable that can hold 40 characters.

`DIM C#(100)` .... Array of 100 double-precision floating-point values

`DIM D$(10) 8` .... Array of 10 fixed-length character strings, each 8 characters in length.

`RDIM E$(4)` .... Array of 4 fixed-length character strings (each can hold 18 characters, or whatever value has been set with the `OPTION LENGTH` instruction).

1-8-4 Local and Global Variables

The BASIC Unit can execute multitasked programs. To enable data transfer among tasks, the Unit is provided with local variables, which can be accessed only within a single task, and global variables, which can be accessed from any task.

Local variables are declared in each task block (between the `PARACT` instruction to the `END PARACT` instruction) and are handled by the task. The same variable name can be used by different tasks; each task has its own, independent copy of the variable.

Global variables are declared outside all task blocks (before the first `PARACT` instruction) and can be used by all tasks. Note that you may not use the same name for a global variable and a local variable.

Example:

`100 DIM ABC(100)` .... Declares a global array

`110 DIM D` .... Declares a global variable

⋮

`200 PARACT 1` .... Beginning of code of task 1

`210 DIM PQR(10)` .... Declares a local array

⋮

`300 X = D + 3` .... Uses global variable `D`. `X` is a local variable.

`310 DIM ABC(3)` .... Error – this local variable name is the same as the global variable declared in line `110`

⋮

`400 END PARACT` .... End of task 1

1-8-5 Non-volatile Variables

The BASIC Unit has battery-backed memory area which does not lose its contents when power is turned off, and which is not initialized when power is turned on. Variables allocated in this area will therefore retain their values even if the BASIC Unit is turned off. These are “non-volatile” variables.

The `RDIM` instruction can be used to force the BASIC Unit to allocate battery-backed memory for use by non-volatile variables. The `RDIM` statement must come before all `PARACT` and `DIM` statements.

Example:

`100 RDIM HOLD(12,5)` .... Declares a non-volatile array variable

`110 RDIM AB` .... Declares a non-volatile variable

`120 DIM P`	Declares a global variable
`130 RDIM Q`	Error – this statement must come before all `DIM` and `PARACT` statements
⋮	
`200 PARACT 1`	Beginning of code of first task
⋮	

Non-volatile memory is cleared by the `OPTION BASE`, `RUN`, and `ERASE` instructions. The contents of non-volatile variables can be saved to a file (memory card of the PC or expansion DM) by the `VSAVE` instruction. The contents can be loaded from a file with the `VLOAD` instruction.

When the current program area is changed with the `PGEN` instruction or a new program is loaded, the non-volatile memory area is not modified. This allows another program to share the same variables. In order to share the variables, however, both programs must use identical `RDIM` statements.

1-9 Type Conversion

The various numeric data types (integer, single-precision floating-point, and double-precision floating-point) will be converted automatically from one type to another when necessary. Character strings are *not* converted automatically. To use a character string containing digits as a number, use the `VAL`, `CVI`, `CVS`, or `CVD` functions; to store a number in a string variable, use the `STR$`, `MKI$`, `MKS$`, or `MKD$` functions.

The following rules are used when the BASIC Unit has to convert from one numeric type to another:

Assignment of Numeric Value to Variable of Different Type

If a numeric value is assigned to a variable of different type, the numeric value is converted into the type of the destination variable.

Example:

`10 LET A% = 1.234`	Assign a single-precision value to an integer variable – value is rounded to 1
`20 PRINT A%`	Print the value of the integer variable

Execution result:

`1`	Variable `A%` contains `1`

Operations on Numeric Values of Different Precisions

If an arithmetic operation is executed between numeric values of different precisions, the precision of the result is dependent on the precisions of the operands and on the operation being performed.

Examples:

`10 PRINT 10%/3%`	Both operands are converted to single-precision floating-point; gives single-precision floating-point answer
`15 PRINT 10%\3%`	Integer division of two integers gives integer answer
`20 PRINT 10%/3!`	Integer operand (`10%`) is converted to single-precision floating-point; gives single-precision floating-point answer
`30 PRINT 10%/3#`	Integer operand (`10%`) is converted to double-precision floating-point; gives double-precision floating-point answer

Execution result:

```
3.33333

3

3.33333

3.3333333333333333
```

Logical Operations

When a logical operation is executed, all the numeric operands are converted into integers.

Example:

```
10 LET A = 1.234 ........... Single-precision value stored in A

20 LET B = NOT A ........... The value of A is converted to integer
                             (1); the NOT operation is performed giv-
                             ing an integer result; the result is con-
                             verted to single-precision and stored in
                             B

30 PRINT B, A .............. Display the values
```

Execution result:

```
–2  1.234 .................. Note that the contents of variable A are
                             not affected by the conversion to integer
                             in line 20
```

Conversion to Integer

When storing a floating-point value in an integer variable, the value is rounded at the first digit below the decimal point.

Example:

```
10 A% = 1.45 ............... Rounded to 1 and stored in A%

20 B% = 1.65 ............... Rounded to 2 and stored in B%

30 PRINT A%, B%
```

Execution result:

```
1  2
```

Conversion into Single-precision Real Number

When a double-precision floating-point value is stored in a single-precision variable, the value is rounded at the seventh digit. When the variable is output to the printer or screen, however, it is rounded at the sixth digit.

Example:

```
10 A! = 3.14159265358 ..... Value rounded at seventh digit;
                            3.141593 is stored in A!

20 PRINT A!
```

Execution result:

```
3.14159 ..................... Single-precision values are rounded at
                              the sixth place for display.
```

1-10 Expressions

An expression is a sequence of constants, variables, and operators that calculate a value which is useful to the programmer. Expressions are divided into numeric, character, relative, and logical types. Examples of each type appear below:

Numeric	Character	Relative	Logical

`20*15/3`
`P*D*(D+4)`
`2.7145`
`A`
`SIN(X)`
	`"BASIC" + "UNIT"`
	`B$ + C$`
	`"BASIC UNIT"`
	`A$`
	`CHR$(31)`
		`A = B`
		`A < B`
		`A <> B`
			`A AND B`
			`A OR B`
			`A XOR B`
			`A < B AND A > C`
			`A = B AND A <> C`

1-10-1 Operators

The BASIC Unit uses three types of operators in its expressions: arithmetic, relative, and logical.

Operators
- Arithmetic operators - - - - Addition, subtraction, multiplication, division, exponentiation, and remainder calculation.
- Relative operators - - - - - Compare two expressions
- Logical operators - - - - - Compound conditions, bit manipulation, and binary operations

Arithmetic Operator

An operator that couples numeric constants and variables and executes arithmetic operations such as addition, subtraction, multiplication, division, exponentiation, and remainder calculation. The available arithmetic operators and their operations are as follows:

Arithmetic operator	Operation	Example	Mathematical notation
+	Addition	`A + B`	A + B
–	Subtraction	`A – B`	A – B
*	Multiplication	`A * B`	A * B
/	Floating-point division	`A / B`	A ÷ B
\	Integer division	`A \ B`	---
^	Exponentiation	`A ^ B`	A^B
`MOD`	Remainder calculation	`A MOD B`	---

- When integer division is executed, the operands are first rounded to integers, then divided; the result is truncated at the decimal point.

Example: integer division

```
123.4\67.89 ──────► 123\68 ────► 1.808 ──────► 1  (result)
           Round to integer      Divide       Truncate
```

- If an attempt is made to divide by zero, an error occurs.
- If an attempt is made to obtain a negative exponent of zero, the result of the operation is not defined, although an error does not occur and operation continues.

Caution If the result of an operation exceeds the range of the result's type, an overflow occurs, but no error is signalled.

Example of integer overflow:

`10 LET A% = 32767` .... No overflow

`20 LET B% = A% + 1` .... An overflow occurs; `B%` contains –32768.

The range of values for each type of numeric value is as follows:

Integer value . . . . –32768 to +32767

Single-precision value . . . . $+3*10^{-39}$ to $+1.7*10^{38}$

Double-precision value . . . . $+1.07*10^{-308}$ to $+1.07*10^{307}$

Relative Operator

Relative operators are used to compare two numeric expressions. They return true (–1) or false (0) depending on the outcome of the comparison. The available relative operators and their operations are as follows:

Relative operator	Operation	Example
=	Equal	`A = B`
`<>, ><`	Not equal	`A <> B, A >< B`
<	Less than	`A < B`
>	Greater than	`A > B`
`<=, =<`	Less than or equal to	`A <= B, A =< B`
`>=, =>`	Greater than or equal to	`A >= B, A => B`

Relative operators are often used in `IF` statements to control the flow of program execution, as follows:

Example: Relative operator in `IF` statement

`IF A <> B THEN 1000` .... If `A` is not equal to `B`, then the program will branch to line `1000`

`IF A$="Y" THEN *PROCESS` ... If `A$` is equal to "`Y`" then the program will branch to label `*PROCESS`

Logical Operator

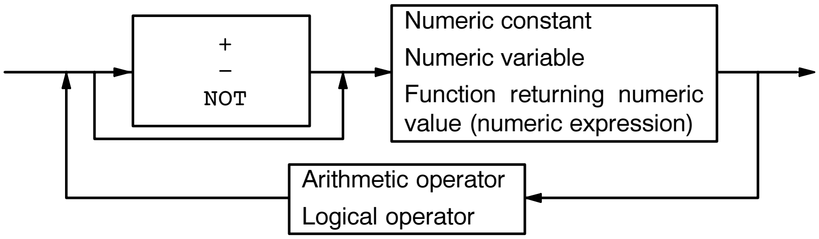
A logical operator performs a logical operation on a single integer value (`NOT`) or between two integer values. Each of the 16 bits in the integer result is set to 0 or 1 based on the result of the logical operation. The available logical operator and their operations are as follows:

Logical operator	Operation	Example
`NOT`	Negation	`NOT A`
`AND`	Logical product	`A AND B`
`OR`	Logical sum	`A OR B`
`XOR`	Logical exclusive sum	`A XOR B`
`IMP`	Implication	`A IMP B`
`EQV`	Equivalence	`A EQV B`

- A logical operator can be used to check the results of two or more relative expressions or to perform bit manipulation (Boolean algebraic operations) on specified numeric values.
- Before a logical operation is executed, the operands are converted to 16-bit, 2's complement integers.

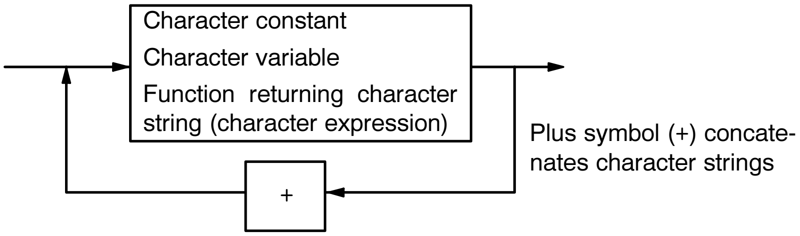
1-10-2 Numeric Expressions

An expression that calculates a numeric value is called a numeric expression. A numeric expression consists of numeric constants, numeric variables, and functions that returns a numeric value, possibly coupled by an arithmetic or logical operator. Parentheses may be used to group parts of a numeric expression.



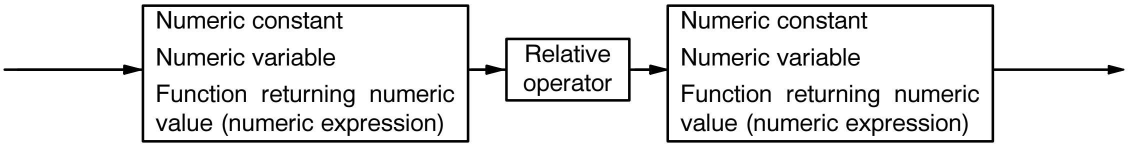
1-10-3 Character Expressions

A character expression returns a character string. These expressions consist of character constants, character variables, and functions that return a character string, coupled by a plus symbol (+). Parentheses may be used to group parts of a character expression.


Character constant
Character variable
Function returning character string (character expression)
+
Plus symbol (+) concatenates character strings

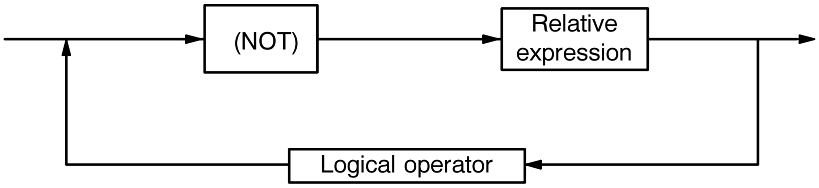
1-10-4 Relative Expressions

A relative expression consists of two numeric expressions coupled by a relative operator.


Numeric constant
Numeric variable
Function returning numeric value (numeric expression)
Relative operator
Numeric constant
Numeric variable
Function returning numeric value (numeric expression)

1-10-5 Logical Expressions

A logical expression consists of two or more numeric or relative expressions coupled by logical operators. These expressions are used to perform bit manipulation and binary operations, and to judge compound conditions. Numeric operands to logical operators are always converted to integers before the operation is performed.



Six types of operators are available for logical expressions: NOT, OR, AND, XOR, IMP, and EQV. The following tables show the results of each operator on single bits A and B:

NOT

A	NOT A
0	1
1	0

AND

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

XOR (Exclusive OR)

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

IMP (Implication)

A	B	A IMP B
0	0	1
0	1	1
1	0	0
1	1	1

EQV (Equivalence)

A	B	A EQV B
0	0	1
0	1	0
1	0	0
1	1	1

Here are some examples of bit manipulation using the logical operators.

Logical Expression: NOT 5 Result: –6

Expression	Integer	Binary
	5	0000000000000101
NOT 5	–6	1111111111111010

Logical Expression: 3 AND 5 Result: 1

Expression	Integer	Binary
	3	0000000000000011
	5	0000000000000101
3 AND 5	1	0000000000000001

Logical Expression: 3 OR 5 Result: 7

Expression	Integer	Binary
	3	0000000000000011
	5	0000000000000101
3 OR 7	7	0000000000000111

Logical Expression: 3 XOR 5 Result: 6

Expression	Integer	Binary
	3	0000000000000011
	5	0000000000000101
3 XOR 5	6	0000000000000110

Logical Expression: 3 IMP 5 Result: –3

Expression	Integer	Binary
	3	0000000000000011
	5	0000000000000101
3 IMP 5	–3	1111111111111101

Logical Expression: 3 EQV 5 Result: –7

Expression	Integer	Binary
	3	0000000000000011
	5	0000000000000101
3 EQV 5	–7	1111111111111001

Logical expressions are often used in `IF` statements to control the flow of program execution according to the result of two or more relative expressions.

Example: Logical expressions in `IF` statements

`IF A>20 AND B>20 THEN 100` . If both `A` and `B` are greater than 20, the program will branch to line `100`.

`IF A>20 OR B>20 THEN 100` . If either `A` or `B` are greater than 20, the program will branch to line `100`.

1-10-6 Functions and System Variables

A function calculates a new value based on an input value (called the function's argument) and returns the new value as its result. A system variable is a special variable whose contents are maintained by the system.

Examples: functions returning a numeric value

`10 LET B = ABS(A)` .... Calculates the absolute value of `A` and stores it in `B`

`20 LET C% = ASC("Hello")` .. Stores the ASCII code for "`H`" in `C%`

`30 LET X# = ATN(ANGLE#)` ... Calculates the arctangent of `ANGLE#` and stores it in `X#`

`40 LET L% = LEN(MY.NAME$)` . Counts the number of characters in `MY.NAME$` and stores the result in `L%`

`50 LET C# = SQR(A^2+B^2)` .. Calculates the square root of A^2+B^2 and stores it in `C#`

Example: Functions returning a character string value

`10 LET A$ = CHR$(34)` ...... Makes a string containing one double-quote (") and stores it in `A$`

`20 PRINT HEX$(A+B)` .... Prints the value of `A+B` in hexadecimal notation

```
30 PRINT "It's now ";TIME$ Prints the current time
```

Note The string returned by character functions is normally limited to 18 characters in length. If you want to use character functions that may return more than 18 characters, use the `DIM` statement to declare a fixed-size character variable that can hold as many characters as you need and assign the result of the function to that variable. For example:

```
40 DIM LONG.RESULT$ 100 ... LONG.RESULT$ is a fixed-length string
                            that can hold up to 100 characters.
   ⋮

320 LET LONG.RESULT$ = LEFT$(SOME.STRING$, 56)
```

1-10-7 Operator Priority

If two or more operators are used in an expression, the BASIC Unit executes the operators starting from the one having the highest priority. If the priorities of the operators are the same, they are executed from the left toward the right. An operator enclosed in ( ) takes precedence over the others. The operators are listed below in order of priority.

Priority	Operator	Operation	Classification
1	`(  )`	Calculates value of expression in parentheses	Expression in ( )
2	Numeric function	Returns numeric value	Function
	Character function	Returns character string	
3	`^`	Exponentiation	Arithmetic operator
4	`–`	Negative sign	
5	`*, /`	Floating-point multiplication and division	
6	`\`	Integer division	
7	`MOD`	Remainder calculation	
8	`+, –`	Addition and subtraction	
9	`=`	Equal to	Relative operator
	`<>, ><`	Not equal to	
	`<, >`	Less than, greater than	
	`<=, =<`	Less than or equal to	
	`>=, =>`	Greater than or equal to	
10	`NOT`	Logical negation	Logical operator
11	`AND`	Logical product	
12	`OR`	Logical sum	
13	`XOR`	Exclusive logical sum	
14	`IMP`	Implication	
15	`EQV`	Equivalence	

SECTION 2
BASIC Instructions

The instructions of the BASIC Unit are broadly classified into commands, statements, functions, and GP-IB instructions.

Commands can be typed in and executed directly from the console in edit or debug mode. Some commands can also be used as statements.

Statements are used in BASIC programs to do most of the program's work and to control the program's execution.

Functions perform a specified calculation and return the result of the calculation to the program. Many functions require one or more arguments.

GP-IB instructions, which control the GP-IB interface, are sub-divided into statements and functions. Note that the GP-IB instructions can be used with Models BSC51 and BSC61 only.

2-1 How to Use this Table

Instruction: This column lists the names of the commands, statements, and functions in alphabetical order.

Syntax: This column describes the form(s) in which the instruction appears in a program, using the following notation:

- Words and symbols in `typewriter font` should be entered exactly as written.
- Items in square brackets ([ ]) may be omitted.
- Items in curly brackets ({ }) indicate choices; alternatives are delimited from each other with the vertical bar character (|). Select one of the alternatives.
- An asterisk (*) indicates that the preceding item or items may be repeated.
- _ indicates a required space. (Spaces can also be used between words and symbols to increase program readability.)
- Words in *italics* are English descriptions of the element that should be substituted. For example, *line-no.* should be replaced with an actual line number.

Purpose: This column presents a brief description of the instruction.

Page: For the further information on the instruction, refer to the page shown in this column.

2-2 Command List

These instructions may be used in EDIT or DEBUG mode. Instructions marked with a diamond (♦) may also be used as statements in programs.

Instruction	Syntax	Purpose	Page
@	@ [*task-number*]	Selects a task to be debugged.	29
AUTO	AUTO [*start-line-no.*] [, *increment*]	Automatically generates line numbers when a program is typed in.	32
BREAK	BREAK [{DELETE {ALL | *line-no.* [, *line-no.*]*} | *line-no.* [, *line-no.*]*}]	Sets, deletes, or lists breakpoints.	33
CLS♦	CLS	Clears screen.	36
CONT	CONT	Resumes execution of program.	37
DELETE	DELETE [*start-line-no.*] [–[*end-line-no.*]]	Deletes program lines.	42
EDIT	EDIT [*line-no.*]	Edits one line of program.	44
FILES / LFILES	FILES [*drive-no.*]	Displays names and size of files in drive.	50
	LFILES [*drive-no.*]	Prints names and sizes of files in drive.	
KILL♦	KILL "*file-name*"	Deletes file.	61
LET♦	[LET] *variable-name* = *expression*	Stores value of *expression* in variable.	62
LIST / LLIST	LIST [*start-line-no.*] [– [*end-line-no.*]]	Displays all or part of program.	64
	LLIST [*start-line-no.*] [– [*end-line-no.*]]	Prints all or part of program.	
LOAD	LOAD "*file-name*"	Reads BASIC program into current program area.	65
MERGE	MERGE "*file-name*"	Reads BASIC program to current program area. Program is merged with any existing program.	68
MON	MON	Sets monitor mode.	71
MSET	MSET [*address*]	Sets upper limit of BASIC program area to allocate machine language program area.	71

Instruction	Syntax	Purpose	Page
NAME♦	NAME "*old-file-name*" AS "*new-file-name*"	Changes file name.	71
NEW	NEW	Deletes program and variables.	71
PGEN	PGEN [*program-no.*]	Selects current program area.	97
PINF	PINF	Displays information on program area.	98
PNAME	PNAME "*program-name*"	Registers or deletes name of current program area.	98
PRINT♦	PRINT [*expression*] [{,|;|␣} [*expression*]]*	Displays value of expression.	99
LPRINT♦	LPRINT [*expression*] [{,|;|␣} [*expression*]]*	Prints value of expression.	
RENUM	RENUM [*new-line-no.*] [, [*old-line-no*.] [, *increment*]]	Re-numbers program lines.	105
ROMLOAD	ROMLOAD	Reads information in EEPROM to user program area.	108
ROMSAVE	ROMSAVE	Writes information in user program area to EEPROM.	109
ROMVERIFY	ROMVERIFY	Verifies between EEPROM and user program area.	109
RUN♦	RUN ["*file-name*"] [, ERASE]	Starts program execution.	109
SAVE	SAVE "*file-name*"	Saves BASIC program to file.	110
STEP	STEP	Executes program one step at a time.	115
TROFF♦	TROFF [{*task-no.*|ALL}]	Stops output of line number trace.	121
TRON♦	TRON [{*task-no.*|ALL}]	Starts output of line number trace.	121
VERIFY	VERIFY "*file-name*"	Verifies program.	125
VLOAD♦	VLOAD "*file-name*"	Reads contents of non-volatile variable from file.	125
VSAVE♦	VSAVE "*file-name*"	Saves contents of non-volatile variable to file.	125
WRITE♦	WRITE *expression* [{,|;|␣}*expression*]*	Displays value of expression.	126

♦The command can also be used as a statement in a program.

2-3 Statement List

Instruction	Syntax	Purpose	Page
ALARM ON / OFF / STOP	ALARM {ON | OFF | STOP}	Enables, disables, or stops time interrupt.	30
BITON / BITOFF	{BITON | BITOFF} *integer-variable*, *bit-position*	Turns ON (1) or OFF (0) the specified bit of an integer variable.	32
CALL	CALL *name* [(*argument* [, *argument*]*)]	Calls a machine language program (subroutine) stored in memory.	33
CLOSE	CLOSE [*#file-no*. [, *#file-no*.]*]	Closes file.	35
CLS	CLS	Clears screen.	36
COM ON / STOP (OFF and STOP are the same)	COM [(*port-no.*)] {ON | STOP}	Enables or stops interrupt from communication line.	36
DATA	DATA *constant* [, *constant*]*	Stores numeric and character constants for use by READ statements.	39
DEF FN	DEF FN*function-name* [(*argument* [, *argument*]*)] = *function-definition-expression*	Defines function.	40

Instruction	Syntax	Purpose	Page
DEG SEG	DEF SEG = *segment-address*	Declares segment address.	42
DEF USR	DEF USR [*no.*] = *start-address*	Defines execution start address of machine language USR function.	42

DEFINT/DEFSNG/ DEFDBL/DEFSTR	{DEFINT | DEFSNG | DEFDBL | DEFSTR}
	{*variable-name* | *character–character*}
	[, {*variable-name* | *character–character*}]*
		Declares variable type.	41

DIM	DIM *variable-name* [(*subscript* [, *subscript*]*)] [*maximum-number-of-characters*] [, *variable-name* [(*subscript* [, *subscript*]*)] [*maximum-number-of-characters*]]*	Declares an array variable or fixed-length string.	42
END	END	Terminates task.	45
END PARACT	END PARACT	Declares the end of a task.	45
ERROR	ERROR *error-no.*	Simulates generation of error.	47
EXIT	EXIT *task-no.*	Terminates specified task.	47
FIELD	FIELD #*file-no.*, *width* AS *character-string-variable* [, *width* AS *character-string-variable*]*	Assigns field variable to random file buffer.	49
FINS ON / STOP (OFF and STOP are the same)	FINS {ON | STOP}	Enables or stops interrupts from network.	50
FOR... TO... STEP... NEXT...	FOR *variable* = *initial-value* TO *final-value* [STEP *increment*] NEXT [*variable* [, *variable*]*]	Repeatedly execute group of statements enclosed by FOR and NEXT statements.	51
GET	GET #*file-no.* [, *record-no.*]	Reads data from random file.	52

GOSUB / RETURN	GOSUB {*line-no.* | *label*}
	RETURN
		Calls subroutine / returns from subroutine.	53

GOTO	GOTO {*line-no.* | *label*}	Branches to specified line or label.	54

IF... THEN...
ELSE...
IF... GOTO...
ELSE...
	IF *conditional-expression*
	THEN {*statement* | *line-no.* | *label*}
	[ELSE {*statement* | *line-no.* | *label*}]
	IF *conditional-expression*
	GOTO {*line-no.* | *label*}
	[ELSE {*statement* | *line-no.* | *label*}]
		Selects statement to be executed according to result of *conditional-expression*.	55

INPUT	INPUT [WAIT *expression*,] ["*prompt*" {, | ;}] *variable* [, *variable*]*	Inputs data to specified variable.	56
INPUT #	INPUT #*file-number*, *variable* [, *variable*]*	Reads data from file into specified variable.	58
KEY ON / OFF / STOP	KEY (*key-no.*) {ON | OFF | STOP}	Enables, disables, or stops interrupts from console numeric keys.	61
KILL	KILL "*file-name*"	Deletes file.	61
LET	[LET] *variable-name* = *expression*	Assigns the value of an expression to a variable	62
LINE INPUT	LINE INPUT [WAIT *expression*,] ["*prompt*" {, | ;}] *character-variable*	Inputs a whole line to a character string variable.	63
LINE INPUT #	LINE INPUT #*file-no.*, *character-variable*	Reads one line from a file into a character string variable.	63
LOCATE	LOCATE *horizontal-position*, *vertical-position*	Moves cursor on screen.	65

Instruction	Syntax	Purpose	Page
LSET/RSET	LSET character-variable = character-expression RSET character-variable = character-expression	Substitutes data into field variable.	67
LPRINT	LPRINT [expression] [{,|;|_} [expression]]*	Prints value of expression.	99
LPRINT USING	LPRINT USING format ; expression [{, | ; | _} [expression]]*	Output value of expression using specified format.	101
MESSAGE	MESSAGE function , message-no.	Allocates and releases message numbers.	68
MID$	MID$(character-expression , expression [, expression]) [= character-expression]	Returns or replaces part of character string variable.	68
NAME	NAME "old-file-name" AS "new-file-name"	Changes file name.	71
ON ALARM GOSUB	ON ALARM time GOSUB {line-no. | label}	Specifies interrupt time and defines interrupt routine.	72
ON COM GOSUB	ON COM [(port-no.)] GOSUB {line-no. | label}	Defines subroutine to process interrupts from communication line.	73
ON ERROR GOTO	ON ERROR GOTO {0 | line-no. | label}	Defines error processing routine and starts error trap.	74
ON FINS GOSUB	ON FINS GOSUB {line-no. | label}	Defines subroutine to process interrupts from network.	74
ON GOSUB	ON expression GOSUB {line-no. | label} [, {line-no. | label}]*	Selects and calls one of several subroutines based on the value of expression.	75
ON GOTO	ON expression GOTO {line-no. | label} [, {line-no. | label}]*	Selects and branches to one of several locations based on the value of expression.	76
ON KEY GOSUB	ON KEY (key-no.) GOSUB {line-no. | label}	Defines subroutine to process numeric key interrupts.	76
ON PC GOSUB	ON PC (interrupt-no.) GOSUB {line-no. | label}	Defines subroutine to process interrupts from PC.	77
ON SIGNAL GOSUB	ON SIGNAL (signal-no.) GOSUB {line-no. | label}	Defines interrupt subroutine for user-defined or system signal.	77
ON TIME$ GOSUB	ON TIME$ = "time" GOSUB {line-no. | label}	Defines subroutine to be executed at a certain time.	78
ON TIMER GOSUB	ON TIMER = interval GOSUB {line-no. | label}	Specifies subroutine to be executed after a certain interval	79
OPEN	OPEN "file-name" [FOR {INPUT | OUTPUT | APPEND}] AS #file-no.	Opens file.	80
OPTION BASE	OPTION BASE {0 | 1}	Declares subscript of first array element.	83
OPTION ERASE	OPTION ERASE	Declares initialization of non-volatile variables.	83
OPTION LENGTH	OPTION LENGTH no.-of-characters	Declares default length for fixed character strings.	84
PARACT	PARACT task-no. [WORK no.-of-bytes]	Declares beginning of task.	85
PAUSE	PAUSE	Stops execution of task until interrupt occurs.	85
PC ON / STOP (OFF and STOP are the same)	PC (interrupt-no.) {ON | STOP}	Enables or stops interrupt from PC.	86

Instruction	Syntax	Purpose	Page
PC READ	PC READ [WAIT time,] "[[#network, node,] source-area, start-word, no.-of-words,] format [, format]*"; variable [, variable]*	Reads data from PC into variable.	86
PC WRITE	PC WRITE [WAIT time,] "[[#network, node,] destination-area, start-word, no.-of-words,] format [, format]*"; variable [, variable]*	Writes value of variable to PC.	93
POKE	POKE address, expression	Writes data to specified address of memory.	99
PRINT	PRINT [expression] [{, | ; | _} [expression]]*	Displays value of expression.	99
PRINT #	PRINT #file-no., [expression] [{, | ; | _} [expression]]*	Outputs value of expression to a file.	99
PRINT USING	PRINT USING format ; expression [{, | ; | _} [expression]]*	Output value of expression in specified format.	101
PRINT # USING	PRINT #file-no., USING format ; expression [{, | ; | _} [expression]]*	Output value of expression in specified format to a file.	101
PUT	PUT #file-no. [, record-no.]	Writes data to random file.	103
RANDOMIZE	RANDOMIZE [expression]	Initializes random series.	103
RDIM	RDIM variable-name [(subscript [, subscript]*)] [maximum-number-of-characters] [, variable-name [(subscript [, subscript]*)] [maximum-number-of-characters]]*	Declares non-volatile variables.	42
READ	READ variable [, variable]*	Reads data from DATA statement and stores it in variable.	104
RECEIVE	RECEIVE message-no., character-variable	Receives message.	105
REM	REM [comment-text]	Causes the BASIC Unit to ignore the comment-text.	105
RESTORE	RESTORE [{line-no. | label}]	Specifies re-use of values in a DATA statement	106
RESUME	RESUME [{0 | line-no. | label | NEXT}]	Exits from error processing routine.	106
RUN	RUN ["file-name"] [, ERASE]	Starts program execution.	109
SEND	SEND message-no., character-expression	Sends message.	111
SENDSIG	SENDSIG signal-no., task-no.	Generates signal.	111
SIGNAL ON / OFF / STOP	SIGNAL signal-no. {ON | OFF | STOP}	Enables, disables, or stops signal interrupt.	112
STOP	STOP	Stops program execution.	115
SWAP	SWAP variable-name, variable-name	Swaps values of two variables.	117
TASK	TASK task-no.	Starts terminated task.	119
TIME$ ON / OFF / STOP	TIME$ {ON | OFF | STOP}	Enables, disables, or stops time interrupt.	120
TIMER ON / OFF / STOP	TIMER {ON | OFF | STOP}	Enables, disables, or stops timer interrupt.	121
TROFF	TROFF [{task-no. | ALL}]	Stops output of line number trace.	121
TRON	TRON [{task-no. | ALL}]	Starts output of line number trace.	121
TWAIT	TWAIT task-no.	Waits for termination of task.	122

Instruction	Syntax	Purpose	Page
VLOAD	VLOAD "*file-name*"	Reads contents of non-volatile variable from file.	125
VSAVE	VSAVE "*file-name*"	Saves contents of non-volatile variable to file.	125
WHILE/WEND	WHILE *conditional-expression* WEND	Repeatedly execute series of statements while condition is satisfied.	126
WRITE	WRITE *expression* [{, | ; | _} [*expression*]]*	Outputs value of expression.	126
WRITE #	WRITE *#file-no.*, *expression* [{, | ; | _} [*expression*]]*	Outputs value of expression to a file.	126

2-4 Function List

Instruction	Syntax	Purpose	Page
ABS	ABS(*expression*)	Calculates the absolute value of the *expression*.	29
ACOS	ACOS(*expression*)	Calculates arc cosine of the *expression*.	29
ASC	ASC(*character-expression*)	Returns the ASCII code of the first character of *character-expression.*	31
ASIN	ASIN(*expression*)	Calculates the arc sine of the *expression*.	31
ATN	ATN(*expression*)	Calculates the arc tangent of the *expression*.	32
CDBL	CDBL(*expression*)	Converts *expression* into a double-precision real number.	33
CHR$	CHR$(*expression*)	Converts *expression* into characters.	34
CINT	CINT(*expression*)	Rounds any fractional part of *expression*	35
COS	COS(*expression*)	Returns cosine of *expression*.	37
CSNG	CSNG(*expression*)	Converts *expression* into single-precision real number.	38

CVI / CVS / CVD	CVI(*2-character-string*)
	CVS(*4-character-string*)
	CVD(*8-character-string*)
		Converts character string into numeric value.	38

DATE$	DATE$ [= "*year/month/day*"]	Returns date of internal clock, or sets date.	40
EOF	EOF(*file-no.*)	Returns true (–1) if *file-no.* has reached end of file; false (0) otherwise.	45
ERL/ERR	ERL ERR	Return line on which error has occurred (ERL) and error code (ERC).	46
EXP	EXP(*expression*)	Calculates exponential function of *expression* ($e^{expression}$)	48
FIX	FIX(*expression*)	Truncates any fractional part of *expression*.	50
FRE	FRE(*expression*)	Returns size of unused memory area.	52
HEX$	HEX$(*expression*)	Returns a character string with the value of *expression* expressed as a hexadecimal number.	54
INKEY$	INKEY$	Returns next character in keyboard buffer.	56
INPUT$	INPUT$(*expression* [, *#file-no.*])	Reads character string of specified length from specified file.	58
INSTR	INSTR([*expression*,] *character-string*, *key-string*)	Searches for *key-string* in *character-string* and returns its position.	59

Instruction	Syntax	Purpose	Page
INT	INT(*expression*)	Returns the largest integer which does not exceed *expression.*	59
INTRB INTRL INTRR	INTRB INTRL INTRR	Variables containing information on an interrupt that has occurred.	60
LEFT$	LEFT$(*character-expression*, *expression*)	Returns the leftmost *expression* characters from *character-expression.*	61
LEN	LEN(*character-expression*)	Returns length of *character-expression.*	62
LOC	LOC(*file-no.*)	Returns current logical position in file.	65
LOF	LOF(*file-no.*)	Returns size of file.	66
LOG	LOG(*expression*)	Calculates natural logarithm of *expression*	66
MID$	MID$(*character-expression*, *length* [, *position*])	Returns *length* characters from *character-expression* starting from *position.*	68
MKI$ / MKS$ / MKD$	MKI$(*integer-value*) MKS$(*single-precision-value*) MKD$(*double-precision-value*)	Converts numeric value into character string.	70
OCT$	OCT$(*expression*)	Returns a character string with the value of *expression* expressed as an octal number.	71
PEEK	PEEK(*address*)	Returns contents of the specified address.	97
RIGHT$	RIGHT$(*character-expression*, *expression*)	Returns the rightmost *expression* characters from *character-expression*	107
RND	RND(*expression*)	Returns random number.	108
SEARCH	SEARCH(*integer-array* [, *expression*] [, *start-element*] [, *increment*])	Searches for first occurrence of the integer value *expression* in *integer-array* and returns element number.	110
SGN	SGN(*expression*)	Returns −1, 0, or 1 depending on whether *expression* is negative, zero, or positive.	112
SIN	SIN(*expression*)	Calculates sine of *expression*.	113
SPACE$	SPACE$(*expression*)	Returns a character string containing *expression* spaces.	113
SPC	SPC(*expression*)	Outputs *expression* spaces.	114
SQR	SQR(*expression*)	Calculates the square root of *expression*.	114
STR$	STR$(*expression*)	Returns a character string with the value of *expression* expressed as a decimal number	116
STRING$	STRING$(*expression*, {*character-string* | *character-code*})	Returns a string with *expression* copies of the first character of *character-expression* or *character-code*.	117
TAB	TAB(*expression*)	Moves cursor to specified column.	118
TAN	TAN(*expression*)	Calculates tangent of *expression*.	119
TIME$	TIME$ [= "*hour*:*minute*:*second*"]	Returns time of internal clock, or sets time.	120
USR	USR[*func-no.*](argument)	Calls a machine language function	122
VAL	VAL(*character-expression*)	Converts *character-expression* into a numeric value.	123
VARPTR	VARPTR(*variable-name*) [, *feature*]	Returns memory address of variable.	123

2-5 GP-IB Instruction List

Statement

Instruction	Syntax	Purpose	Page
CMD DELIM	CMD DELIM = delimiter-code	Specifies delimiter.	128
CMD PPR	CMD PPR = mode	Selects PPR mode.	128
CMD TIMEOUT	CMD TIMEOUT = timeout-parameter	Specifies limit value for timeout check.	128
INPUT @	INPUT@ [talker-address [, listener-address [, listener-address]*]]; variable [, variable]*	Receives data sent from specified talker and stores it in variable.	129
IRESET REN	IRESET REN	Makes REN (remote enable) false.	129
ISET IFC	ISET IFC [, integer]	Transmits IFC (interface clear).	129
ISET REN	ISET REN	Makes REN (remote enable) true.	130
ISET SRQ	ISET SRQ [@] [N]	Transmits SRQ (service request).	130
LINE INPUT @	LINE INPUT@ [talker-address [, listener-address [, listener-address]*]]; character-string-variable	Receives string data sent from specified talker and substitutes it into character string variable.	130
ON SRQ GOSUB	ON SRQ GOSUB {line-no. | label}	Specifies first line of SRQ subroutine.	131
POLL	POLL talker-address, numeric-variable [; talker-address, numeric-variable]*	Performs serial polling.	131
PPOLL	PPOLL [PPU] [, listener-address, integer]*	Assigns response output line for parallel polling.	132
PRINT @	PRINT@ [listener-address [, listener-address]*]; [data [,data]*] [@]	Transmits data as ASCII character string.	132
RBYTE	RBYTE [command] [, command]*; [numeric-variable [,numeric-variable]*	Receives binary data after transmitting multi-line message.	133
SRQ ON/OFF/STOP	SRQ {ON | OFF | STOP}	Controls reception of SRQ.	133
WBYTE	WBYTE [command] [, command]*; [data [, data]*] [@]	Transmits multi-line message and binary data.	133

Function

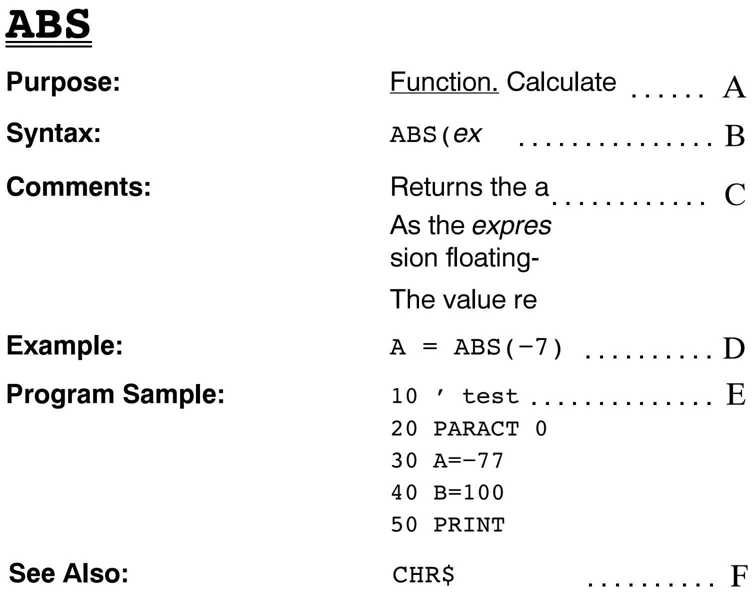
Instruction	Syntax	Purpose	Page
IEEE(0)	IEEE(0)	Checks the delimiter.	134
IEEE(1)	IEEE(1)	Checks the initialized status of GP-IB interface.	134
IEEE(2)	IEEE(2)	Checks the talker and listener status, and received interface message.	134
IEEE(4)	IEEE(4)	Stores the device status of the device that transmits the service request during serial polling.	134
IEEE(5)	IEEE(5)	Stores the talker address of the device that transmits the service request during serial polling.	134
IEEE(6)	IEEE(6)	Stores the talker address of the device that does not respond to the serial polling.	134
IEEE(7)	IEEE(7)	Stores the data byte obtained as a result of parallel polling.	135
STATUS	STATUS	Stores device status.	135

SECTION 3
Instruction Reference

This section describes the commands, statements, and functions of the BASIC Unit. The instructions are listed in alphabetical order. GP-IB instructions are listed separately.

3-1 Guide to Reference

Instructions are listed alphabetically. The description of each instruction is as follows:



ABS

Purpose:	Function. Calculate
Syntax:	`ABS(`*ex*
Comments:	Returns the a As the *expres* sion floating- The value re
Example:	`A = ABS(−7)`

Program Sample:	`10 ' test`
	`20 PARACT 0`
	`30 A=−77`
	`40 B=100`
	`50 PRINT`

See Also:	`CHR$`

1. **Purpose:**

   Indicates whether the instruction is a command, statement, or function and describes its basic purpose.

2. **Syntax:**

   Describes the form(s) in which the instruction appears in a program, using the following notation:.

   - Words and symbols in `typewriter font` should be entered exactly as written.
   - Items in square brackets ([ ]) may be omitted.
   - Items in curly brackets ({ }) indicate choices; alternatives are delimited from each other with the vertical bar character (|). Select one of the alternatives.
   - An asterisk (*) indicates that the preceding item or group of items may be repeated.
   - _ indicates a required space. (Spaces can also be used between words and symbols to increase program readability.)
   - Words in *italics* are English descriptions of the element that should be substituted. For example, *line-no.* should be replaced with an actual line number.

3. **Comments:**

   Describes any information pertinent to the instruction, such as the range of its arguments, and any points to be noted.

4. **Example:**

   Describes a simple example of the instruction's usage.

5. **Program Sample:**

   Shows a sample program or program fragment using the instruction.

6. **See Also:**

   Indicates other instructions with related features.

3-2 Reference

@

Purpose: Command. Selects a task to be referred to.

Syntax: `@[` *task-no.* `]`

Comments: Selects the task specified by the *task-no.* for referral when `BREAK` or `STOP` is executed.

When this command is used during execution of a `BREAK` or `STOP`, references will be changed to the task specified by the *task-no.* and the local variables in the task can be referenced by direct execution of a `PRINT` statement. If *task-no.* is omitted, the number of the task currently being referred to is displayed. If this command is executed after the program has been executed, a message is output indicating the task number and the last line number executed in the task.

Example:

```
@3
```

ABS

Purpose: Function. Calculates the absolute value of the *expression*.

Syntax: `ABS(`*expression*`)`

Comments: Returns the absolute value of the specified *expression*.

An integer, single-precision floating-point, or double-precision floating-point expression can be specified as the *expression*.

The value returned is of the same type as *expression*.

Example:

```
A = ABS(-7)
```

Program Sample:

```
10 ’ test command name     :ABS
20 PARACT 0
30 A=-77
40 B=100
50 PRINT ”ABS(”;A;”)-->”;ABS(A)
60 PRINT ”ABS(”;B;”)-->”;ABS(B)
70 END
80 END PARACT
Ok
RUN
ABS(-77 )--> 77
ABS( 100 )--> 100
Ok
```

ACOS

Purpose: Function. Calculates the arc cosine of the *expression*.

Syntax: `ACOS(`*expression*`)`

Comments: Returns the arc cosine of the specified *expression*. Specify the *expression* in radians.

If the *expression* is of type integer or single-precision, then single-precision is assumed and the result of the calculation is of type single-precision. If the *expression* is of type double-precision, then double-precision is assumed and the result is of type double-precision.

Example:

```
A = ACOS(0.4)
```

Program Sample:

```
10 ’test command name :ACOS
20 PARACT 0
30 PRINT ”Calculates the arc cosine.”
40 INPUT ”Input numeric value...”,VALUE
50 ANGLE=180/3.141592*ACOS(VALUE)
60 PRINT ”Arc cosine is”;ANGLE;”degree.”
70 END
80 END PARACT
Ok
RUN
Calculates the arc cosine.
Input numeric value... 0.5
Arc cosine is 60 degree.
```

See Also: ASIN, ATN

ALARM ON/OFF/STOP

Purpose: Statement. Enables, disables, or stops time interrupt.

Syntax: ALARM { ON | OFF | STOP }

Comments: ON enables the interrupt. When this statement is executed, the program execution branches to a defined processing routine if an interrupt occurs.

OFF disables the interrupt. When this statement is executed, the program execution does not branch to a defined processing routine even if an interrupt occurs.

STOP disables the interrupt. When this statement is executed, the program execution does not branch to a defined processing routine even if an interrupt occurs. However, the occurrence of the interrupt is recorded and the execution branches to the defined routine when the interrupt is enabled.

Note

1. Interrupts are stopped immediately after the ON ALARM GOSUB statement is executed.
2. Only one interrupt is allowed for each ON ALARM GOSUB statement. Interrupts are stopped when execution has branched to the interrupt processing routine.

Example: ALARM ON

Program Sample:

```
10 ’ test command name :ALARM
20 PARACT 0
30 CLS
40 ON ALARM 300 GOSUB *ALON
50 ’
60 ALARM ON
70 LOCATE 0,0:PRINRT ”Alarm timer set TIME”;TIME$
80 PAUSE
90 END
100 *ALON
110 LOCATE 0,5:PRINT ”Alarm signal occurs”;SPACE$(12);TIME$
120 RETURN
130 END PARACT
```

See Also: ON ALARM GOSUB

ASC

Purpose: Function. Returns the character code of the first character in *character-string*.

Syntax: ASC(*character-string*)

Comments: Calculates the character code of the first character of *character-string* and returns it as an integer.

Example: A = ASC(B$)

Program Sample:

```
10 ’ test command name           :ASC
20 PARACT 0
30 ’Converts ”a” and ”A” into their character codes
40 SMALL$=”a”
50 S=ASC(SMALL$)
60 L=S–32
70 LARGE$=CHR$(L)
80 PRINT SMALL$;”––>”;S
90 PRINT LARGE$;”––>”;L
100 END
110 END PARACT
Ok
RUN
a––-> 97
A––-> 65
```

See Also: CHR$

ASIN

Purpose: Function. Calculates the arc sine of the *expression*.

Syntax: ASIN(*expression*)

Comments: Returns the arc sine of the specified *expression*. Specify *expression* in radians.

If the *expression* is of type integer or single-precision, then single-precision is assumed and the result of the calculation is of type single-precision. If the *expression* is of type double-precision, then double-precision is assumed and the result is of type double-precision.

Example: A = ASIN(0,3)

Program Sample:

```
10 ’test command name :ASIN
20 PARACT 0
30 PRINT ”Calculates the arc sine.”
40 INPUT ”Input numeric value...”,VALUE
50 ANGLE=180/3.141592*ASIN(VALUE)
60 PRINT ”Arc sine is”;ANGLE;”degree.”
70 END
80 END PARACT
Ok
RUN
Calculates the arc sine.
Input numeric value... 0.5
Arc sine is 30 degree.
```

See Also: ACOS, ATN

ATN

Purpose: Function. Calculates the arc tangent of the *expression*.

Syntax: `ATN (`*expression*`)`

Comments: Returns the arc tangent of the specified *expression*. Specify the *expression* in radians.

If the *expression* is of type integer or single-precision, then single-precision is assumed and the result of the calculation is of type single-precision. If the *expression* is of type double-precision, then double-precision is assumed and the result is of type double-precision.

Example: `A = ATN(1)`

Program Sample:

```
10 'test command name :ATN              '
20 PARACT 0
30 PRINT "Calculates arc tangent."
40 P#=3.14159265359
50 INPUT "Input numeric value...",A
60 X=ATN(A);Y=X/P#*180
70 PRINT "ATN(X) =";Y;"degree."
80 END
90 END PARACT
Ok
RUN
Calculates the arc tangent.
Input numeric value... 0.5
ATN(X) = 26.5651 degree.
```

See Also: `ACOS`, `ASIN`

AUTO

Purpose: Command. Automatically generates line numbers when a program is input.

Syntax: `AUTO` [*start-line-no.*] [,*increment*]

Comments: Automatically generates line numbers starting from *start-line-no.* in increments of *increment*.

Both *start-line-no.* and *increment* can be set in the range 1 to 65529. The default for both is 10.

`AUTO` is canceled when the abort key is pressed or when a non-existent *start-line-no.* is input. If a line number that exists is input, only that line is deleted.

Note If a line number that already exists is generated, that line number is displayed. The cursor is moved to the end of statement + 1 column, allowing editing using the `EDIT` command.

Example: `AUTO 1000,20`

See Also: `EDIT`

BITON/BITOFF

Purpose: Statement. Turns ON (1) or OFF (0) the specified bit of an integer variable.

Syntax: {`BITON` | `BITOFF`} *integer-variable*`,` *bit-position*

Comments: Specify the *bit-position* as an integer from 0 to 15. The least significant bit is 0.

Example: `BITON N,3`

Program Sample:

```
10 ’test command name :BIT ON/OFF       ’
20 PARACT 0
30 A%=0
40 BITON A%,4
50 PRINT ”BITON : ”;A%
60 BITOFF A%,4
70 PRINT ”BITOFF : ”;A%
80 END
90 END PARACT
Ok
RUN
BITON  :  16
BITOFF :  0
```

BREAK

Purpose: Command. Sets, deletes, or lists breakpoints.

Syntax: BREAK [{DELETE {ALL | *line-no.* [, *line-no.*]*} | *line-no.* [, *line-no.*]*}]

Comments: Sets or deletes a breakpoint on a line specified by *line-no.*. Up to 10 breakpoints can be set or deleted.

If BREAK is used without arguments all the breakpoints currently set are listed.

Each breakpoint stops the program execution before the line on which the breakpoint is set is to be executed, and displays the following message:

Break in *line-no.*

Example: BREAK 120

CALL

Purpose: Statement. Calls and executes a machine language program (subroutine) stored in memory.

Syntax: CALL *name* [(*argument* [,*argument*]*)]

Comments: The *name* must be an integer variable and its value must be the offset of the execution start address of the machine language program to be called.

The execution branches to an address indicated by the segment specified by the DEF SEG statement executed immediately before and by the offset specified by the *name*.

Specify any variables to be passed to the machine language program as *arguments*. Variables of any type can be specified as *arguments* but constants and expressions cannot be used.

From the machine language program called by the CALL statement, control can be returned to the calling program by the machine language RETF instruction.

Example: CALL AB%(N,A$)

See Also: DEF SEG, USR

CDBL

Purpose: Function. Converts *expression* into a double-precision floating-point number.

Syntax: CDBL(*expression*)

Comments: Converts the integer or single-precision floating-point number *expression* into a double-precision floating-point number.

The *expression* can be double-precision floating-point type. In this case CDBL has no effect.

Example:

```
C# = CDBL(B!/4)
```

Program Sample:

```
10 ’test command name :CDBL             ’
20 PARACT 0
30 A!=123.45678!
40 B#=CDBL(A!)
50 PRINT ”A=”;A!, ”CDBL(A)=”;B#
60 END
70 END PARACT
Ok
RUN
A= 123.457   CDBL(A)= 123.4567794799805
```

See Also: CINT

CHR$

Purpose: Function. Converts *expression* into a character.

Syntax: CHR$(*expression*)

Comments: Returns the character whose character code is equal to *expression*. Specify the *expression* in the range 0 to 255.

Example:

```
A$ = CHR$(67)
```

Program Sample:

```
10 ’test command name   :CHR$        ’
20 PARACT 0
30 ’ Outputs character codes 48 through 69”
40 FOR I= 48 TO 69
50             PRINT ”[”;I;”]--->”;CHR$(I)
60 NEXT I
70 END
80 END PARACT
Ok
RUN
[ 48 ]--->0
[ 49 ]--->1
[ 50 ]--->2
[ 51 ]--->3
[ 52 ]--->4
[ 53 ]--->5
[ 54 ]--->6
[ 55 ]--->7
[ 56 ]--->8
[ 57 ]--->9
[ 58 ]--->:
[ 59 ]--->;
[ 60 ]---><
[ 61 ]--->=
[ 62 ]--->>
[ 63 ]--->?
[ 64 ]--->@
[ 65 ]--->A
[ 66 ]--->B
[ 67 ]--->C
[ 68 ]--->D
[ 69 ]--->E
```

See Also: ASC

CINT

Purpose: Function. Converts *expression* into an integer.

Syntax: `CINT(`*expression*`)`

Comments: Converts the single-precision floating-point number or double-precision floating-point number specified by *expression* into an integer.

The *expression* can be of integer type.

Specify *expression* in the range –32768 to 32767. Fractions are rounded up if positive, down if negative.

Example:

```
A% = CINT(B#)
```

Program Sample:

```
10 ’test command name :CINT             ’
20 PARACT 0
30 A=75.57
40 B=–5.51
50 C%=CINT(A)
60 D%=CINT(B)
70 PRINT ”A=”;A,”CINT(A)=”;C%
80 PRINT ”B=”;B,”CINT(B)=”;D%
90 END
100 END PARACT
Ok
RUN
A= 75.57      CINT(A)= 76
B=–5.51       CINT(B)=–6
```

See Also: `CDBL`, `CSNG`, `FIX`, `INT`

CLOSE

Purpose: Statement. Closes file.

Syntax: `CLOSE` [*#file-no*. [, *#file-no*.]*]

Comments: Closes the file specified by *file-no*. The *file-no*. is the number assigned to the file when the file was opened.

Once a file is closed the file number is no longer associated with it.
A file number previously assigned to a closed file can be used when opening the same or a different file. A closed file can be opened by specifying the same or a different file number.

If *file-no.* is omitted, all open files are closed.

`CLOSE` dumps all the data in the buffer when the file was opened for output. To correctly terminate output processing of the file, `CLOSE` must be executed.

Note Files are closed automatically by `END` and `RUN`.

Example:

```
CLOSE #1
```

Program Sample:

```
10 ’test command name :CLOSE
20 PARACT 0
30 OPEN ”INDATA” FOR OUTPUT AS #1
40 PRINT #1 ”BASIC UNIT”
50 ’Closes specified file no.
60 CLOSE #1
70 OPEN ”INDATA” FOR INPUT AS #1
80 LINE INPUT #1,L$
90 OPEN ”OUTDATA” FOR OUTPUT AS #2
100 PRINT #2,L$
```

```
110 CLOSE #2
120 OPEN ”OUTDATA” FOR INPUT AS #2
130 LINE INPUT #2,K$
140 ’
150   PRINT K$
160 ’
170 CLOSE
180 END
190 END PARACT
Ok
RUN
BASIC UNIT
```

See Also: END, OPEN, RUN

CLS

Purpose: Command or Statement. Clears screen.

Syntax: `CLS`

Comments: Clears the screen and returns the cursor to the home position.

Example: `CLS`

Program Sample:

```
10 ’test command name :CLS
20 PARACT 0
30 INPUT ”Input X”;X
40 INPUT ”Input Y”;Y
50 IF X > Y THEN PRINT ”X is greater than Y”
60 IF Y > X THEN PRINT ”X is less than Y”
70 IF X = Y THEN PRINT ”X is equal to Y”
80 FOR I=0 TO 1000 :NEXT I
90 CLS
100 END
110 END PARACT
Ok
RUN
Input X? 12
Input Y? 13
X is less than Y
```

COM ON/OFF/STOP

Purpose: Statement. Enables, disables, or stops interrupt from communication line.

Syntax: COM [(*port-no.*)] {ON | OFF | STOP}

Comments: Specify port 1 to 3 as the *port-no*. (Port 1 is the default)

ON enables the interrupt so that program execution branches to a defined processing routine if an interrupt occurs.

OFF operates the same as STOP.

STOP stops the interrupt. When this statement is executed, the program execution does not branch to the processing routine even if an interrupt occurs. However, the occurrence of the interrupt is recorded and execution branches to the processing routine after the interrupt is enabled.

Note

1. The interrupt is stopped immediately after the ON COM GOSUB statement has been executed.
2. The interrupt is stopped when control has been passed to the interrupt processing routine.

Example: `COM (1) ON`

See Also: `ON COM GOSUB`

CONT

Purpose: Command. Resumes execution of the program.

Syntax: `CONT`

Comments: Resumes execution of a program halted by the `STOP` statement, `BREAK` command, or abort key.

`Quit in` *line-no* will be displayed when the program is stopped with CTRL + X, in which case `STEP` or `CONT` cannot be used. If `BREAK in` *line-no* is displayed when the program is stopped with CTRL + C, it is possible to use `STEP` or `CONT`.

Note When the program is halted and modified, it cannot be resumed by `CONT`. In addition, the program may not be able to be resumed depending on the timing at which its execution has been halted.

Example: `CONT`

See Also: `STOP, BREAK`

COS

Purpose: Function. Returns the cosine of *expression*.

Syntax: `COS(`*expression*`)`

Comments: Returns the cosine of the specified *expression*. Specify the *expression* in radians.

If the *expression* is of type integer or single-precision, then single-precision is assumed and the result of the calculation is of type single-precision. If the *expression* is of type double-precision, then double-precision is assumed and the result is of type double-precision.

Example: `A = COS(`*ANGLE*`)`

Program Sample:

```
10 ’test command name :COS              ’
20 PARACT 0
30 FOR X=0 TO 180 STEP 10
40           C=COS(X/180*3.1415927#)
50           PRINT ”Angle:”;X, ”Cosine:”;C
60 NEXT X
70 END
80 END PARACT
Ok
RUN
Angle: 0     Cosine: 1
Angle: 10    Cosine: .984808
Angle: 20    Cosine: .939693
Angle: 30    Cosine: .866025
Angle: 40    Cosine: .766044
Angle: 50    Cosine: .642788
Angle: 60    Cosine: .5
Angle: 70    Cosine: .34202
Angle: 80    Cosine: .173648
Angle: 90    Cosine:-2.32051E-08
Angle: 100   Cosine:-.173648
Angle: 110   Cosine:-.34202
```

```
Angle: 120   Cosine:-.5
Angle: 130   Cosine:-.672788
Angle: 140   Cosine:-.766044
Angle: 150   Cosine:-.866025
Angle: 160   Cosine:-.939693
Angle: 170   Cosine:-.984808
Angle: 180   Cosine:-1
```

See Also: SIN, TAN

CSNG

Purpose: Function. Converts *expression* into a single-precision floating-point number.

Syntax: CSNG(*expression*)

Comments: Converts an integer or double-precision floating-point number *expression* into a single-precision floating-point number. The *expression* can be single-precision floating-point type.

Specify *expression* in the range +3.4*10^{-38}.

Example: A! = CSNG(B#/2)

Program Sample:

```
10 ’test command name :CSNG             ’
20 PARACT 0
30 D#=9876.5432#
40 S=CSNG(D#)
50 PRINT ”Double-precision floating-point number”;TAB(28);
55 PRINT ”Single-precision floating-point number”
60 PRINT D#;TAB(28);S
70 END
80 END PARACT
Ok
RUN
Double-precision floating-point number  Single-precision
floating-point number
 9876.5432                       9876.54
```

See Also: CINT

CVI/CVS/CVD

Purpose: Function. Converts character string into numeric value.

Syntax:

CVI(*2-character-string*)
CVS(*4-character-string*)
CVD(*8-character-string*)

Comments: Restores the numeric value converted by MKI$/MKS$/MKD$.

CVI converts the 2-character string converted by MKI$ into an integer.

CVS converts the 4-character string converted by MKS$ into a single-precision floating-point number.

CVD converts the 8-character string converted by MKD$ into a double-precision floating-point number.

Example: A = CVI(B$)

Program Sample:

```
10 ’test command name      :CVI/CVS/CVD ’
20 PARACT 0
30 OPEN ”DATA1” AS #1
40 FIELD #1,4 AS TESTI$,4 AS TESTS$,8 AS TESTD$
50 LSET TESTI$=MKI$(100%)
```

```
60 LSET TESTS$=MKS$(1.23456)
70 LSET TESTD$=MKD$(12345.123456789#)
80 PUT #1,1
90 ’
100 ’
110 GET #1,1
120 I%=CVI(TESTI$)
130 S!=CVS(TESTS$)
140 D#=CVD(TESTD$)
150 PRINT “CVI-->”;I%
160 PRINT “CVS-->”;S!
170 PRINT “CVD-->”;D#
180 CLOSE
190 END
200 END PARACT
Ok
RUN
CVI--> 100
CVS--> 1.23456
CVD--> 12345.123456789
```

See Also: MKI$/MKS$/MKD$

DATA

Purpose: Statement. Stores numeric constants and character constants to be read by READ statements.

Syntax: DATA *constant* [, *constant*]*

Comments: The DATA statement is a non-executable statement and can be placed anywhere in the task block of the program. As many DATA statements as required can be placed in one program.

The *constant* can be a character constant, integer constant, single-precision floating point constant, or double-precision floating point constant.

If the data of the DATA statement is a character constant and is regarded as a numeric value, the character constant must be enclosed with a pair of double quotation marks (”).

Example: DATA 3,AB,CDE,7,1357

Program Sample:

```
10 ’test command name :DATA
20 PARACT 0
30 READ NAME1$,NAME2$
40 READ NUMB1,NUMB2,NUMB3
50 PRINT NAME1$,NAME2$
60 PRINT NUMB1,NUMB2,NUMB3
70 END
80 DATA BASIC,UNIT
90 DATA 3.14,1.41,1991
100 END PARACT
Ok
RUN
BASIC UNIT
3.14         1.41         1991
```

See Also: READ, RESTORE

DATE$

Purpose: Function. Returns date of internal clock, or sets date.

Syntax: `DATE$` [= "*year/month/day*"]

Comments: Returns the current date if ="*year/month/day*" is omitted. The value returned is of character type. If ="*year/month/day*" is specified, the date is set.

The format of the date is as follows:

```
"YY/MM/DD"   YY: year (lower 2 digits)
             MM: month (01 to 12)
             DD: day (01 to 31)
```

Example: `A $ = DATE$`

Program Sample:

```
10 'test command name :DATE$            '
20 PARACT 0
30 YY$=LEFT$(DATE$,2)
40 MM$=MID$(DATE$,4,2)
50 DD$=RIGHT$(DATE$,2)
60 PRINT "TODAY IS ";YY$;".";MM$;".";DD$;"."
70 END
80 END PARACT
Ok
RUN
TODAY IS 92.06.04.
```

See Also: `TIME$`

DEF FN

Purpose: Statement. Defines a function.

Syntax: `DEF` *function-name* `[(`*argument* `[,` *argument*]*`)]` = *function-definition-expression*

Comments: The *function-name* must start with `FN` followed by, as the third character, a letter of the alphabet.

The type of result returned by the third character of the function's name. For example, a function called `FNX` would return an integer if the `DEFINT X` statement had been executed, or a double-precision value if `DEFDBL X` had been executed.

The *arguments* correspond to variables having the same names in *function-definition-expression*. These variables are valid only when the *function-definition-expression* is evaluated.

A variable having the same name as that of an *argument* can be used in the program.

The `DEF FN` statement must be used before using the function in a task.

The *function-name* must be unique in any one task.

Note The *function-name* defined by the `DEF FN` statement is valid only in the same task.

Example: `DEF FNS(A,B) = A*16+B`

Program Sample:

```
10 'test command name :DEF FN
20 PARACT 0
30 DEF FNS(A,H)=A*H/2
40 PRINT "Calculates the area of triangle."
50 INPUT "Base = ";A
```

```
60 INPUT ”Height = ”;H
70 PRINT ”Area = ”;FNS(A,H)
80 END
90 END PARACT
Ok
RUN
Calculates the area of triangle.
Base = ? 10
Height = ? 20
Area = 100
```

DEFINT/DEFSNG/DEFDBL/DEFSTR

Purpose: Statement. Declares variable type.

Syntax: {DEFINT | DEFSNG | DEFDBL | DEFSTR} {*character* | *character–character*} [, {*character* | *character–character*}]*

Comments: The DEF statement declares the type of a single variable or a set of variables beginning with *character*.

The type declared by a type declarator (%, !, #, or $) takes precedence over the type declared by the DEF statement.

A range of variables can be specified by linking characters with a hyphen.

DEFINT declares integer type, DEFSNG declares single-precision floating point type, DEFDBL declares double-precision floating point type, and DEFSTR declares character type.

Note This declaration must be made before declaring or using all variables, and therefore, is valid in all tasks.

Example:

```
DEFINT I–N
```

Program Sample:

```
10 ’test command name :DEFINT
20 DEFINT I – N
30 DEFSNG S
40 DEFDBL D
50 DEFSTR A
60 PARACT 0
70 I=3.14159265358979#
80 S=3.14159265358979#
90 D=3.14159265358979#
100 A=”Ratio of the circumference”
110 PRINT SPC(8);A
120 PRINT ”Integer ”;I
130 PRINT ”Single-precision ”;S
140 PRINT ”Double-precision ”;D
150 END
160 END PARACT
Ok
RUN
Ratio of the circumference
Integer  3
Single-precision  3.14159
Double-precision  3.14159265358979
```

DEF SEG

Purpose: Statement. Declares segment address.

Syntax: DEF SEG = *segment-address*

Comments: Defines a segment address for use when a memory address is specified by the CALL, DEF USR, or POKE statement, or by PEEK or USR function.

Unless this declaration is made, the segment address is 0.

Example: DEF SEG = &H0100

See Also: CALL, DEF USR, POKE, PEEK, USR, VARPTR

DEF USR

Purpose: Statement. Defines the execution start address of the machine language program for a USR function.

Syntax: DEF USR[*no.*] = *start-address*

Comments: The *no.* is 0 to 9. This number is used to identify a USR function when two or more USR functions are used.

0 is the default if *no.* is omitted.

As the *start-address*, specify the offset of the execution start address of the machine language program to be called.

As the segment value, the value of the DEF SEG statement executed immediately before is used.

Example: DEF USR3 = &H0800.

See Also: DEF SEG, USR

DELETE

Purpose: Command. Deletes program lines.

Syntax: DELETE {*start-line-no.* [–] | – *end-line-no.*}

Comments: Either a *start-line-no.* or an *end-line-no.* must be specified.

If *start-line-no.* only is specified, then only that line is deleted.

If *start-line-no.* is specified followed by a hyphen (–), all the lines following and including the start line number are deleted.

If the *end-line-no.* is specified preceded by a hyphen, all the lines starting from the first line to (and including) the specified end line are deleted.

Example: DELETE 500–999

DIM/RDIM

Purpose: Statement. Declares a variable. If an array variable, declares the number of dimensions and the maximum value of the subscript of each dimension. If the variable is a character string, the maximum number of characters can be declared.
The DIM statement declares ordinary global and local variables.
The RDIM statement declares non-volatile variables.

Syntax: RDIM *variable-name* [(*subscript* [, *subscript*]*)] [*maximum-number-of-characters*] [, *variable-name* [(*subscript* [, *subscript*]*)] [*maximum-number-of-characters*]]*

Comments: The number of dimensions and the maximum number of subscripts of all the array variables must be declared before the variables are referenced. When the DIM statement is used to declare a local variable, make the declaration immediately after the PARACT statement.

The RDIM statement can make a declaration only before a task. When using the DIM statement at the same time, the RDIM statement must come before the DIM statement. The RDIM and DIM statements cannot be used to make a multi-statement in a single line.

The *subscript* must be a constant.

The lower-limit value of the *subscript* can be set to be 0 or 1 by the OPTION BASE statement.

The *maximum-number-of-characters* can be specified for a character string variable.

The maximum value of *maximum-number-of-characters* is 538. Specify a numeric constant.

If *maximum-number-of-characters* is omitted, the character length specified by the OPTION LENGTH statement is assumed.

If neither *maximum-number-of-characters* nor the OPTION LENGTH statement is specified, the maximum number of characters of the character string variable is 18.

A simple, local variable can be used without being declared by the DIM statement.

A local variable must not have the same name and type as a global variable.

Note An array is like a table of values. Each element in the table is accessed by giving the table's name and the element number in parentheses. In the examples below, it is assumed that the lower limit on array subscripts is set to its default value, 0. Thus, an array declared as DIM X(2) would have 3 elements: X(0), X(1), and X(2).

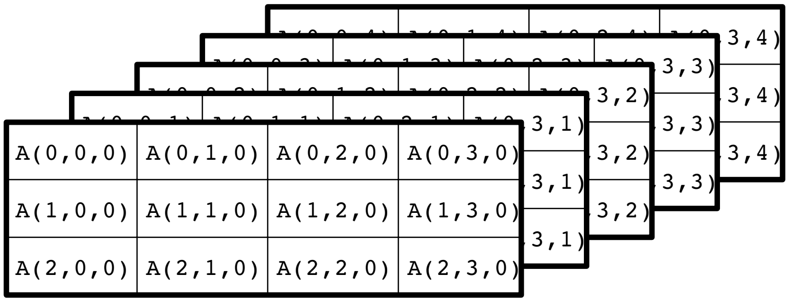
DIM A(2): 1-dimensional array, 3 elements

A(0)	A(1)	A(2)

DIM A(2,3): 2-dimensional array, 12 elements

A(0,0)	A(0,1)	A(0,2)	A(0,3)
A(1,0)	A(1,1)	A(1,2)	A(1,3)
A(2,0)	A(2,1)	A(2,2)	A(2,3)

DIM A(2,3,4): 3-dimensional array, 60 elements



Example:

```
DIM AZ(100)
```

Program Sample:

```
10 'test command name :DIM
20 PARACT 0
30 DIM A(3,5)
40 FOR I=1 TO 3
50     FOR J=1 TO 5
60             A(I,J)=I*J
70     NEXT J
80 NEXT I
90 FOR I= 1 TO 3
```

```
100    FOR J=1 TO 5
110          PRINT "A(";I;",";J;") =";A(I,J)
120    NEXT J
130 NEXT I
140 END
150 END PARACT
Ok
RUN
A(1,1) = 1
A(1,2) = 2
A(1,3) = 3
A(1,4) = 4
A(1,5) = 5
A(2,1) = 2
A(2,2) = 4
A(2,3) = 6
A(2,4) = 8
A(2,5) = 10
A(3,1) = 3
A(3,2) = 6
A(3,3) = 9
A(3,4) = 12
A(3,5) = 15
```

See Also: `OPTION BASE`, `OPTION ERASE`, `OPTION LENGTH`

EDIT

Purpose: Command. Edits one line of the program.

Syntax: `EDIT [`*line-no.*`]`

Comments: Displays the specified program line and allows editing by inserting or deleting characters and moving the cursor.

If *line-no.* is omitted, the line executed last is selected.

If the line specified by *line-no.* does not exist, an error occurs.

The following edit keys can be used:

Right key	Moves the cursor to the right. This key has no effect when the cursor is at the last column of the last line.
Left key	Moves the cursor to the left. This key has not effect when the cursor is at the beginning of the first line.
Up key	Moves the cursor up one line. When this key is pressed with the cursor on the top line, the cursor moves to the beginning of the line. With the cursor at this position, this key has no effect.
Down key	Moves the cursor down one line. If this key is pressed with the cursor at the bottom line, the cursor moves to the last column of the line. With the cursor at this position, this key has no effect.
HOME	Moves the cursor to the beginning of the first line.
BS or **DEL**	Deletes the character to the left of the cursor. This key has no effect when the cursor is at the beginning of a line.
RETURN	Carriage return.
CTRL + E	Deletes the characters to the left of the cursor.
CTRL + L or **CLR**	Clears the entire screen. The cursor moves to the home position.
CTRL + R or **INS**	Switches between typeover and insert modes. The default mode is typeover.

Note Some keys do not operate depending on the console used.

The BS and DEL keys have the same effect on the FIT Terminal Pack, but for consoles, whether the character at the cursor position is deleted by the DEL key or not depends on the console.

Example: `EDIT 120`

END

Purpose: Statement. Terminates task.

Syntax: `END`

Comments: Terminates the execution of a task. All the files currently open are closed.

Even if the `END` statement is missing, the task is terminated when the `END PARACT` statement is executed.

Example: `END`

Program Sample:

```
10 'test command name :END
20 PARACT 0
30 X=3 :Y=5
40 PRINT "3 * 5 =";X*Y
50 END
60 PRINT "No execution after END"
70 END PARACT
Ok
RUN 3 * 5 = 15
Ok
```

See Also: `END PARACT`

END PARACT

Purpose: Statement. Declares the end of a task.

Syntax: `END PARACT`

Comments: When this statement is executed, the task is terminated. All the files currently opened are closed.

All tasks must end with this statement (and must begin with a corresponding `PARACT` statement).

This statement cannot be part of a multi-statement.

Example: `END PARACT`

See Also: `END`, `PARACT`

EOF

Purpose: Function. Returns true at the end of a sequential file.

Syntax: `EOF (`*file-no.*`)`

Comments: The *file-no.* must be that of a file opened in input mode. When the file ends, –1 (true) is returned; otherwise, 0 (false) is returned.

If the specified file is a communication port, and if the input buffer is empty, –1 (true) is returned.

Example: `EOF (3)`

Program Sample:

```
10 'test command name :EOF              '
20 PARACT 0
30 OPEN "DATA" FOR OUTPUT AS #1
```

```
40 FOR I=1 TO 10
50     PRINT #1,I*I
60 NEXT I
70 CLOSE
80 OPEN ”DATA” FOR INPUT AS #1
90 IF EOF (1) THEN 130
100    INPUT #1,A$
110    PRINT A$
120 GOTO 90
130 CLOSE
140 END
150 END PARACT
Ok
RUN
1
4
9
16
25
36
49
64
81
100
```

See Also: END, PARACT

ERL/ERR

Purpose: Function. Returns line on which error has occurred and also error code.

Syntax:

```
ERL
ERR
```

Example:

```
A = ERL
```

Comments: The ERL function returns the number of the line on which an error has occurred. The ERR function returns the error code of the error.

These functions retain the error number until the next error occurs.

Program Sample:

```
10 ’test command name :ERL/ERR          ’
20 PARACT 0
30 ON ERROR GOTO *ERRPRO
40 FOR I=1 TO 5
50     READ A
60 NEXT I
70 DATA 1,2,3,4
80 END
90 ’
100 *ERRPRO
110 PRINT ”ERROR NO. IS”;ERR;”.”
120 PRINT ”ERROR LINE IS”;ERL;”.”
130 RESUME 80
140 END PARACT
Ok
RUN
ERROR NO. IS 4 .
ERROR LINE IS 50 .
```

ERROR

Purpose: Statement. Simulates the generation of an error.

Syntax: `ERROR` *error-no.*

Comments: Generates an error specified by *error-no.*

Specify *error-no.* in the range 0 to 255.

When `ERROR` is executed, the values which will be returned by `ERR` and `ERL` are updated. I.e., the `ERR` function will return the error number and the `ERL` function will return the error line.

When the `ON ERROR GOTO` statement exists, program execution branches to the specified line number or label.

Example:

```
ERROR 128
```

Program Sample:

```
10 'test command name :ERROR
20 PARACT 0
30 ON ERROR GOTO *ERRPRO
40 INPUT "1: Error generation test 9: Execution ends";ER
50 IF ER=1 THEN ERROR 255
60 IF ER=9 THEN END ELSE GOTO 40
70 '
80 *ERRPRO
90 IF ERR=255 THEN PRINT "Error has occurred."
100 RESUME NEXT
110 END PARACT
Ok
RUN
1: Error generation test 9: Execution ends? 1
Error has occurred.
1: Error generation test 9: Execution ends? 1
Error has occurred.
1: Error generation test 9: Execution ends? 9
Ok
```

See Also: `ERR/ERL`, `ON ERROR GOTO`, `RESUME`

EXIT

Purpose: Statement. Terminates specified task.

Syntax: `EXIT` *task-no.*

Comments: Terminates execution of the task specified by *task-no.*

Use the number specified by the `PARACT` statement as the *task-no.*

A task from which execution has `EXIT`ed can be resumed by the `TASK` statement.

An error occurs if a *task-no.* that does not exist in the program is specified.

Example:

```
EXIT 5
```

Program Sample:

```
10 'test command name :EXIT
20 PARACT 0
30 PRINT "TASK0 START"
40 PRINT
50 TASK 1
55 GOTO 60   'WAIT
60 EXIT 1
70 PRINT:PRINT
```

```
80 PRINT "TASK1 STOP"
90 PRINT:PRINT
100 TASK 1
105 GOTO 110 'WAIT
110 EXIT 1
120 TASK 1
125 GOTO 130 'WAIT
130 EXIT 1
140 END
150 END PARACT
160 PARACT 1
170 PRINT "TASK1"
180 GOTO 170
190 END
200 END PARACT
Ok
RUN
TASK0 START

TASK1


TASK1 STOP


TASK1
TASK1
Ok
```

See Also: PARACT, TASK

EXP

Purpose: Function. Calculates the exponential function of *expression* with base e.

Syntax: EXP(*expression*)

Comments: If the *expression* is of type integer or single-precision, then single-precision is assumed and the result of the calculation is of type single-precision. If the *expression* is of type double-precision, then double-precision is assumed and the result is of type double-precision.

Example: EXP(1)

Program Sample:

```
10 'test command name :EXP              '
20 PARACT 0
30 CLS
40 FOR X=0 TO 4 STEP 0.2
50           E=EXP(X)
60           F=FIX(E)
70           PRINT SPC(F);"*"
80 NEXT X
90 END
100 END PARACT
Ok
RUN
 *
 *
 *
 *
```

```
  *
  *
   *
   *
      *
       *
         *
           *
              *
                    *
                           *
                                  *
                                        *
                                                      *
```

FIELD

Purpose: Statement. Assigns a character string variable to a random file buffer.

Syntax: `FIELD` *#file-no.*`,` *width* `AS` *character-string-variable*
`[,` *width* `AS` *character-string-variable*]*

Comments: Assigns the *character-string-variable*, of width *width*, to the random file specified by *file-no.*

It is possible to specify two or more `FIELD` statements in different formats to one *file-no.*, but only the `FIELD` statement immediately preceding is valid for the `PUT` and `GET` statements.

Specify *width* in the range 1 to 256.

If two or more *widths* are specified, ensure that the total number of characters are 256 or less.

If the character length of the *character-string-variable*, (i.e., *width*) is specified as more than the character length specified by the `OPTION LENGTH` statement, `RDIM` statement, or `DIM` statement, an error occurs. If the `OPTION LENGTH`, `RDIM`, or `DIM` statement is not used and if the *width* is specified to be longer than 18 characters, an error also occurs.

Substitute a value for the *character-string-variable* using `LSET` or `RSET`. If the number of characters to be substituted is less than the *width*, the character string is left-justified when the `LSET` statement is used, or right-justified when the `RSET` statement is used, and the remaining space is filled with blanks. If the substitution is made without using the `LSET` or `RSET` statement, the positions of the characters exceeding the *width* are filled with undefined values.

Note An error occurs if `FIELD` is executed on a file which has not been opened in random mode.

Example:

```
FIELD # 1,128 AS B$
```

Program Sample:

```
10 ’test command name :FIELD
20 PARACT 0
30 OPEN ”INVENTORY” AS #1
40 FIELD #1,18 AS PNAME$, 2 AS NINVNT$
50 INPUT ”PRODUCT NAME: ”;PN$
60 INPUT ”INVENTORY: ”;NI%
70 LSET PNAME$=PN$
80 LSET NINVENT$=MKI$(IN%)
90 PUT #1, 1
100 GET #1, 1
110 PRINT ”PRODUCT NAME: ”;PNAME$,”INVENTORY: ”;CVI(NINVNT$)
120 CLOSE
```

```
130 END
140 END PARACT
Ok
RUN
PRODUCT NAME :? BASIC
INVENT :? 100
PRODUCT NAME: BASIC             INVENTORY: 100
```

See Also: `GET, RSET, LSET, PUT`

FINS ON/OFF/STOP

Purpose: Statement. Enables, disables, or stops interrupt from network.

Syntax: `FINS {ON | OFF | STOP}`

Comments: `ON` enables the interrupt. When this statement is executed and if an interrupt occurs, the program execution branches to a defined processing routine.

`STOP` stops the interrupt. When this statement is executed, the program execution does not branch to the defined routine when an interrupt occurs. However, the occurrence of the interrupt is recorded, and the program execution branches to the defined processing routine when the interrupt is later enabled.

`OFF` operates the same as `STOP`.

Note

1. The interrupt is stopped immediately after `ON FINS GOSUB` has been executed.
2. The interrupt is stopped when the control is passed to an interrupt processing routine.

Example:

```
FINS ON
```

See Also: `ON FINS GOSUB`

FILES/LFILES

Purpose: Command. Displays file names and sizes on a specified drive.

Syntax: `FILES [`*drive-no.*`]`
`LFILES [`*drive-no.*`]`

Comments: Specify 0 or 1 as the *drive-no.* If *drive-no.* is omitted, defaults to 0.

The `LFILES` command prints the file names and sizes instead of displaying them on the screen.

Example:

```
FILES
```

FIX

Purpose: Function. Returns the truncated whole-number portion of *expression.*

Syntax: `FIX (`*expression*`)`

Comments: Returns the value of the *expression* truncated at the decimal point.

An integer, single-precision floating point, or double-precision floating point expression can be specified.

If *expression* is of integer or single-precision floating point type, the result is of single-precision floating point type. If it is of double-precision floating point type, the result is also of double-precision floating point type.

Example:

```
A = FIX(B/3)
```

Program Sample:

```
10 'test command name :FIX                    '
20 PARACT 0
```

```
30 INPUT ”INPUT Floating point number.”
40 INPUT X
50 Y=FIX(X)
60 Z=INT(X)
70 PRINT ”FIX(”;X;”)=”;Y
80 PRINT ”INT(”;X;”)=”;Z
90 END
100 END PARACT
Ok
RUN
INPUT Floating point number.
? –1.3
FIX( –1.3 )=–1
INT( –1.3 )=–2
```

See Also: INT

FOR TO STEP/NEXT

Purpose: Statement. Allows repeat execution of a group of statements enclosed within FOR and NEXT statements a specified number of times.

Syntax: FOR *variable* = *initial-value* TO *final-value* [STEP *increment*]
NEXT [*variable* [, *variable*]*]

Comments: The FOR statement specifies the beginning of a loop and the NEXT statement specifies the end of the loop. The statements in the loop are repeatedly executed a specified number of times.

Only a simple numeric variable can be used for *variable*.

The *variable* is used as a counter and is incremented by the *increment* each time the loop is executed.

If the *increment* is 0, the loop is executed forever.

If the STEP statement is omitted, the *increment* is assumed to be 1.

If *final-value* is greater than *initial-value*, then *variable* is initialized to *initial-value* and the statements in the loop are repeatedly executed while the value of *variable* does not exceed *final-value*.

If the *final-value* is less than *initial-value*, the *variable* is initialized to *initial-value* and the statements in the loop are repeatedly executed until the value of *variable* falls below the *final-value*. (In this case, *increment* must be a negative number. If *increment* is greater than 0, the loop is not executed at all.)

The program execution can jump into or out of a FOR loop using the GOTO statement.

The *variable* of the FOR statement must be the same as the *variable* of the NEXT statement. The *variable* of the NEXT statement need not be specified.

The NEXT statement can correspond to more than one FOR statement when more than one *variable* is listed.

An error results when the number of nestings exceeds 100.

Example:

```
FOR N = 1 TO 100
NEXT N
```

Program Sample:

```
10 ’test command name :FOR NEXT
20 PARACT 0
30 CLS
40 FOR I=10 TO 69 STEP 2
50          FOR J=20 TO 0 STEP –2
60                 LOCATE I,J:PRINT ”*”;
```

```
70            NEXT J
80 NEXT I
90 END
100 END PARACT
```

See Also: WHILE/WEND

FRE

Purpose: Function. Returns the size of unused memory in the specified area.

Syntax: FRE (*expression*)

Comments: Returns the size of the unused memory area specified by the value of *expression*.

Valid *expression* values are:

0: S code area (stores user BASIC program and machine language program)

1: RDIM area (stores non-volatile variables)

2: Memory area (variable storage area + E code (execution code) storage area)

The value returned is double-precision floating point type.

Example: PRINT FRE(0)

Program Sample:

```
10 'test command name :FRE
20 PARACT 0
30 PRINT "Unused memory area is as follows:"
40 PRINT "S code area ";FRE(0);"bytes free"
50 PRINT "RDIM area   ";FRE(1);"bytes free"
60 PRINT "Memory area ";FRE(2);"bytes free"
70 END
100 END PARACT
Ok
RUN
Unused memory area is as follows:
S code area  65343 bytes free
RDIM area    32512 bytes free
Memory area  151427 bytes free
```

GET

Purpose: Statement. Reads data from the random file specified.

Syntax: GET *#file-no.* [, *record-no.*]

Comments: If *record-no.* is omitted, the record number of the immediately preceding PUT or GET statement is used. If neither the PUT nor the GET statement has been executed, 0 is used.

The minimum value of *record-no.* is 1, and the maximum value is 32767.

Note

1. This statement causes an error if executed to a file which has not been opened in random mode.
2. Record length of records specified by *record-no.* is 256 bytes.

Example: GET #3

Program Sample:

```
10 'test command name :GET#
20 PARACT 0
30 OPEN "INVENTORY" AS #1
40 FIELD #1,18 AS PNAME$,2 AS NINVNT$
```

```
50 'CREATES THE INVENTORY FILE.
60 LSET PNAME$="BASIC"
70 RSET NINVNT$=MKI$(100)
80 PUT #1,1
90 '
100 'READS THE INVENTORY FILE
110 FOR I=1 TO LOF(1)
120          GET #1,I
130          PRINT "PRODUCT NAME: ";SNAME$,"INVENTORY: ";
135 PRINT CVI(NINVNT$)
140 NEXT I
150 CLOSE
160 END
170 END PARACT
Ok
RUN
PRODUCT NAME: BASIC        INVENTORY: 100
```

See Also: FIELD, OPEN, PUT

GOSUB/RETURN

Purpose: Statement. Calls a subroutine and returns from it.

Syntax:

GOSUB {*line-no.* | *label*}
[RETURN]

Comments: The GOSUB statement calls the subroutine specified by the *line-no.* or **label*. When the RETURN statement in the subroutine is executed, the execution returns to the statement immediately after the GOSUB.

The RETURN statement can be used more than once in a subroutine.

The execution does not exit from a subroutine unless the subroutine has a RETURN statement.

Note A subroutine of a different task cannot be called.

Example:

```
GOSUB *LABEL4
RETURN
```

Program Sample:

```
10 'test command name :GOSUB
20 PARACT 0
30 'SELECT PROCESS, SET EACH SUBROUTINE
40 CLS
50 *MAINROUTINE
60 LOCATE 10,10 :INPUT "1:INPUT 2:CALCULATION 3:END :";A$
70 IF A$="1" THEN GOSUB *SUB1
80 IF A$="2" THEN GOSUB *SUB2
90 IF A$="9" THEN END
100 GOTO *MAINROUTINE
110 '
120 *SUB1
130 LOCATE 10,12 :PRINT SPACE$(25)
140 LOCATE 10,13 :PRINT SPACE$(25)
150 LOCATE 10,12 :INPUT " UNIT PRICE OF PARTS :";UNIT
160 LOCATE 10,13 :INPUT " DELIVERED QUANTITY :";QNTTY
170 RETURN
180 '
190 *SUB2
200 LOCATE 10,15 :PRINT SPACE$(25)
210 LOCATE 10,15 :PRINT " PRICE :";UNIT*QNTTY
```

```
220 RETURN
230 END PARACT
Ok
RUN
1:INPUT 2:CALCULATION 3:END :? 1

UNIT PRICE OF PARTS :? 20
DELIVERED QUANTITY  :? 20
1:INPUT 2:CALCULATION 3:END :? 2

PRICE : 400
1:INPUT 2:CALCULATION 3:END :? 9
Ok
```

See Also: GOTO, ON GOSUB

GOTO

Purpose: Statement. Unconditionally branches to the specified line.

Syntax: GOTO {*line-no.* | *label}

Comments: Branches the program execution to the line specified by *line-no.* or *label.

Note Execution cannot be branched to other task blocks.

Example: GOTO 2000

Program Sample:

```
10 ’test command name :GOTO
20 PARACT 0
30 CLS
40 START
50 PRINT ”WELCOME TO BASIC, HIT ANY KEY.”
60 A$=INKEY$ :IF A$=”” THEN GOTO *START
70 INPUT ”EXECUTE AGAIN (Y/N)”;ANS$
80 IF AN$=”Y” OR ANS$=”y” THEN GOTO 30
90 END
100 END PARACT
```

See Also: GOSUB, ON GOTO

HEX$

Purpose: Function. Converts *expression* into a hexadecimal character string.

Syntax: HEX$(*expression*)

Comments: Converts the *expression* into a character string representation of the hexadecimal value.

Fractional *expressions* are rounded before the conversion.

The *expression* must be in the range –32768 to 65535. The value returned ranges from 0 to FFFF.

Example: A$ = HEX$(49)

Program Sample:

```
10 ’test command name :HEX$
20 PARACT 0
30 PRINT ”DECIMAL”,”HEXADECIMAL”
40 FOR X=1 TO 16
50   PRINT X,”&H”;HEX$(X)
60 NEXT X
70 END
```

```
80 END PARACT
Ok
RUN
DECIMAL       HEXADECIMAL
1             &H1
2             &H2
3             &H3
4             &H4
5             &H5
6             &H6
7             &H7
8             &H8
9             &H9
10            &HA
11            &HB
12            &HC
13            &HD
14            &HE
15            &HF
16            &H10
```

IF THEN ELSE

Purpose:

Statement. Selects statement to be executed according to result of *conditional-expression*.

Syntax:

IF *conditional-expression* THEN {*statement* | *line-no.* | *label*}
[ELSE {*statement* | *line-no.* | *label*}]
IF *conditional-expression* GOTO {*line-no.* | *label*}
[ELSE {*statement* | *line-no.* | *label*}]

Comments:

Evaluates *conditional-expression* and executes the *statement* following THEN if the value of the *conditional-expression* is true (non-zero). If *conditional-expression* evaluates to false (zero), the *statement* following ELSE is executed.

If a *line-no.* or *label* is given instead of a *statement*, the program will branch to that line or label.

Note Any input expression used in *conditional-expression* must be assigned in advance. For example, the first line below must be rewriten to the second line.

Wrong: `TIMER ON : IF A$=INPUT$(1) THEN...`
Correct: `TIMER ON : B$=INPUT$(1) : IF A$=B$ THEN...`

Example:

```
IF A = 6 THEN PRINT ABC ELSE 3000
```

Program Sample:

```
10 ’test command name :IF THEN
20 PARACT 0
30 CLS
40 INPUT ”Input X”;X
50 INPUT ”Input Y”;Y
60 IF X > Y THEN PRINT ”X is greater than Y”
70 IF Y > X THEN PRINT ”X is less than Y”
80 IF X = Y THEN PRINT ”X is equal to Y”
90 END
100 END PARACT
Ok
RUN
Input X? 13
Input Y? 14
X is less than Y
```

INKEY $

Purpose: Function. Returns the next character in the keyboard buffer.

Syntax:

```
INKEY$
```

Comments: If there are any characters in the keyboard buffer, the first one is returned. If there are no characters in the buffer, a null string is returned. (The program will *not* wait until a key is pressed.)

Example:

```
IF INKEY$ = "" GOTO 1000
```

Program Sample:

```
10 'test command name :INKEY$                  '
20 PARACT 0
30 PRINT "Input key. (end: space)"
40 *KEYIN
50 K$ = INKEY$ : IF K$ = "" GOTO 50
60 PRINT "You pressed " ; K$
70 IF K$ = CHR$(32) THEN END ELSE *KEYIN
80 END PARACT
```

INPUT

Purpose: Statement. Inputs data to specified variable.

Syntax:

INPUT [WAIT *specified time*,] ["*prompt*" {, | ;}] *variable* [, *variable*]*
INPUT *#file-no.*, *variable* [, *variable*]*

Comments: When the first form is executed, the *prompt* (if any) is displayed and the system waits for input from the terminal. If a semicolon (;) is specified after the *prompt*, a question mark and a space are displayed after the *prompt*. If a comma (,) is specified, the question mark is not displayed.

The value entered from the terminal is stored in the *variable* when the return key is pressed.

If more than one *variable* is specified, the user must respond with the same number of values. Each value must be separated from the others with a comma (,).

When reading a character string from the terminal, leading and trailing spaces are normally removed from the answer before the string is stored. In order to input a string with leading or trailing spaces, or a string containing a comma, the user must type the string between a pair of double quotes ("). In this case, no quotation marks can be included in the character string.

If the type of the *variable* does not match the type of the input value, or if the number of *variables* does not match the number of values, the message "?Redo from start" is displayed, and the system waits for the input.

If WAIT is specified, and the input of data is not completed within the specified time, the INPUT statement causes an error. Time is specified in units of 0.1 second; the value of the time expression must be between 1 and 32767. To wait until the input is completed, either specify a *time* of 0, or do not use WAIT at all.

The INPUT# statement reads data from the file opened as *file-no*. The data in the file must match the type necessary for the INPUT statement. In addition, the number of values necessary for the INPUT statement must exist in the file.

If the file specified by *file-no.* is a terminal, the effect is the same as the INPUT statement.

Note

1. To use the INPUT# statement, the file must have been opened in INPUT mode.
2. Interruption is possible while an input instruction is being executed. When input is interrupted with a definition such as ON..., the program jumps to

the defined destination for execution. Refer to *Section 6 Advanced Programming* of the *BASIC Unit Operation Manual* for details. When an interruption is being executed the source of the interruption is the STOP status.

Example:

```
INPUT WAIT 100, "KEY" ; A$
```

Program Sample:

```
10 'test command name :INPUT
20 PARACT 0
30 'Converts year of Showa into A.C.
40 INPUT "Year of Showa [1-64]  : ";YS
50 IF YS<1 OR YS>64 THEN PRINT "INPUT ERROR" : GOTO 40
60 AC=YS+1925
70 PRINT "A.C.";AC
80 END
90 END PARACT
Ok
RUN
Year of Showa [1-64] : ? 16
A.C. 1941

10 'test command name :INPUT#
20 PARACT 0
30 OPEN "DATA1" DOR OUTPUT AS #1
40 PRINT #1, "DATA 1"
50 PRINT #1, "DATA 2"
60 PRINT #1, "DATA END"
70 CLOSE
80 '
90 OPEN "DATA1" FOR INPUT AS #1
100 *SINPUT
110   IF EOF(1) GOTO 150
120   INPUT #1,AS
130   PRINT A$
140 GOTO *SINPUT
150 CLOSE
160 END
170 END PARACT
Ok
RUN
DATA 1
DATA 2
DATA END

10 'test command name :INPUT WAIT
20 OPTION LENGTH 26
30 PARACT 0
40 ALP1$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
50 ALP2$="abcdefghijklmnopqrstuvwxyz"
60 PRINT "Input alphabet (A-Z) in 10 seconds"
70 INPUT WAIT 100, INP$
80 '
90 IF INP$=ALP1$ OR INP$=ALP2$ THEN PRINT "PASSED!" :GOTO 110
100 PRINT "Typing error"
110 END
120 END PARACT
```

See Also:

```
LINE INPUT, OPEN
```

INPUT $

Purpose: Function. Reads a character string of the specified length from a file.

Syntax: `INPUT$(`*expression* `[, `*#file-no.*`])`

Comments: Reads *expression* characters the file specified by *file-no.* (The file must have been opened in `INPUT` mode.)

If *file-no.* is omitted, `INPUT$` reads from the terminal. In this case, characters typed are not shown on the screen

The system waits until the number of characters specified by the *expression* have been input.

Example:

```
A$ = INPUT$(10,#3)
```

Program Sample:

```
10 'test command name :INPUT$
20 PARACT 0
30 OPEN "DATA" FOR OUTPUT AS #1
40 FOR I=1 TO 10
50   PRINT #1,"ABCDEF";
60 NEXT I
70 CLOSE
80 PRINT "OUTPUTS 10 CHARACTERS AT A TIME"
90 PRINT "HIT ANY KEY":A$=INPUT$(1)
100 OPEN "DATA" FOR INPUT AS #1
110 IF EOF(1) GOTO 160
120   DA$ = INPUT$(10,#1)
130   PRINT DA$
140 PRINT "HIT ANY KEY":A$ =INPUT$(1)
150 GOTO 110
160 CLOSE
170 END
180 END PARACT
Ok
RUN
OUTPUTS 10 CHARACTERS AT A TIME
HIT ANY KEY

ABCDEFABCD
HIT ANY KEY

EFABCDEFAB
HIT ANY KEY

CDEFABCDEF
HIT ANY KEY

ABCDEFABCD
HIT ANY KEY

EFABCDEFAB
HIT ANY KEY

CDEFABCDEF
HIT ANY KEY

Ok
```

Note Interruption is possible while an input instruction is being executed. When input is interrupted with a definition such as `ON`..., the program jumps to the defined destination for execution. After the execution, the next line will be executed. When an interruption is being executed, the same interruption is in the STOP status until RETURN, so the program will not jump again to the top of what will be executed. Other interruption causes must be stopped using STOP (TIMER STOP) before using INPUT$. Refer to *Section 6 Advanced Programming* of the *BASIC Unit Operation Manual* for details.

INSTR

Purpose: Function. Searches *character-expression* for *key-string* and returns its position.

Syntax: INSTR([*expression*,] *character-expression*, *key-string*)

Comments: Searches the character string which is the value of *character-expression* to find the *key-string*. If *key-string* is found, INSTR returns the position of the first character of the *key-string*; otherwise, it returns 0.

If *expression* is specified, the search starts from *expression* characters past the start of the *character-expression*. If *expression* is omitted, 1 is assumed, and the search starts from the beginning of the *character-expression*.

If the *character-expression* is a null string (""), 0 is returned.

If the *key-string* is a null string ("") the value of *expression* is returned (or 1, if *expression* is omitted).

If the *expression* is greater than the length of the *character-expression*, 0 is returned.

Example:

```
A = INSTR(2,B$,"KEY")
```

Program Sample:

```
10 'test command name :INSTR                   '
20 PARACT 0
30 DIM A$25
40 A$="OMRON, OA, CAD/CAM, FA, NC"
50 PRINT "CHARACTER STRING=";A$
60 NTH=INSTR(1,A$,"FA")
70 PRINT "FA starts at the";NTH;“th character of the string."
80 END
90 END PARACT
Ok
RUN
CHARACTER STRING=OMRON, OA, CAD/CAM, FA, NC
FA starts at the 18th character of the string.
```

INT

Purpose: Function. Returns the largest integer which does not exceed *expression*.

Syntax: INT(*expression*)

Comments: *Expression* can be an integer or single- or double-precision floating point value. If the *expression* is of integer or single-precision type, the result is a single-precision floating point value. If the *expression* is of double-precision type, the result is also a double-precision floating point value.

INT and FIX return the same value for positive arguments, but the value returned by INT for negative arguments is one less than the value returned by FIX.

Example:

```
A = INT(B)
```

Program Sample:

```
10 'test command name :INT                     '
20 PARACT 0
```

```
30 X=−1.41
40 Y=3.14
50 PRINT ”When X = ”;X;”and Y = ”;Y;”,”
60 Z=X+Y
70 I=INT(X)+INT(Y)
80 PRINT ”X + Y =”;Z
90 PRINT ”INT(X)+INT(Y)=”;I
100 END
110 END PARACT
Ok
RUN
When X = −1.41 and  Y = 3.14,
X + Y = 1.73
INT(X)+INT(Y)= 1
```

See Also: CINT, FIX

INTRB/INTRL/INTRR

Purpose: Function. System variables containing information on an interrupt that has occurred.

Syntax:

```
INTRB
INTRL
INTRR
```

Comments: The values of INTRB and INTRR are set only in an interrupt routine. Outside of interrupt routines, their values are 0. Therefore, substitute the values in an interrupt routine and refer to the values in the main routine.

INTRB contains the number of the line that set up the interrupt processing routine.

INTRL contains the number of the line that was executing when the interrupt occurred, or 0 if no line was executing.

INTRR contains a number indicating the interrupt source.

When an interrupt occurs, the current values of INTRB and INTRR are saved and the new values are stored; when an interrupt routine returns, the previous values of INTRB and INTRR are restored. These variables, therefore, contain the correct values throughout the execution of the interrupt routine, even if nested interrupts occur.

Interrupt source	INTRR value
User-defined signal (1 to 5)	1 through 5
COM (1 to 3)	6 through 8
Signal (STOP)	10
Signal (PC watchdog timer error)	11
Signal (cyclic error)	12
Signal (battery error)	13
Alarm	14
Timer	15
Time	16
SRQ	17
FINS	18
KEY (0 through 9)	20 through 29
PC (1 through 15)	31 through 45

Note The program cannot change the values of INTRB, INTRL, or INTRR.

Example: A = INTRB

See Also: ERR/ERL, ON {ALARM | FINS | KEY | SIGNAL | TIME$ | TIMER} GOSUB

KEY ON/OFF/STOP

Purpose: Statement. Enables, disables, or stops interrupts from numeric keypad.

Syntax: KEY (*key-no.*) {ON | OFF | STOP}

Comments: *Key-no.* is the number of a numeric key (from 0 to 9).

ON enables the interrupt. After this statement is executed, program execution will branch to the interrupt routine when the key is pressed.

OFF disables the interrupt. After this statement is executed, program execution will not branch to the interrupt routine even if the key is pressed.

STOP stops the interrupt. After this statement is executed, the program execution does not branch to the interrupt routine when an interrupt occurs, but the occurrence of the interrupt is recorded, and the interrupt routine will be called when the interrupt is re-enabled.

Note The interrupt is turned OFF immediately after the execution of an ON KEY GOSUB statement.

Key interrupts are stopped during the execution of the key interrupt routine.

Example: KEY(7) ON

See Also: ON KEY GOSUB

KILL

Purpose: Command or Statement. Deletes file.

Syntax: KILL "*file-name*"

Comments: Deletes the file specified by the *file-name*. If the file name has an extension, the extension must be included in *file-name*.

For details on the *file-name*, refer to the description of the OPEN statement.

Note An error occurs if the specified file is open.

Example: KILL "0:TEST1.BAS"

Program Sample:

```
10 'test command name :KILL
20 PARACT 0
30 OPEN "TEMP1" FOR OUTPUT AS #1
40 CLOSE #1
50 NAME "TEMP1" AS "TEMP2"
60 KILL "TEMP2"
70 END
80 END PARACT
Ok
RUN
Ok
```

LEFT $

Purpose: Function. Returns the leftmost *expression* characters from *character-expression*.

Syntax: LEFT$(*character-expression*, *expression*)

Comments: Returns a character string of the length specified by *expression* from the left of *character-expression*.

If the *expression* is 0, a null string is returned.

If the value of *expression* exceeds the length of the *character-expression*, the entire *character-expression* is returned.

Example:

```
A$ = LEFT$(B$,4)
```

Program Sample:

```
10 ’test command name :LEFT$                    ’
20 PARACT 0
30 YY$=LEFT$(DATE$,2)
40 MM$=MID$(DATE$,4,2)
50 DD$=MID$(DATE$,7,2)
60 PRINT ”Today is ’”;YY$”.”;MM$;”.”;DD$;”.”
70 END
80 END PARACT
Ok
RUN
Today is ’92.07.04.
```

See Also: `MID$`, `RIGHT$`

LEN

Purpose: <u>Function.</u> Returns length of *character-expression*.

Syntax: `LEN(`*character-expression*`)`

Comments: Counts the characters in *character-expression* and returns the number counted.

Non-printing characters and blanks are included in the count.

Example:

```
A = LEN(B$)
```

Program Sample:

```
10 ’test command name :LEN                      ’
20 PARACT 0
30 INPUT ”Input character string”;A$
40 L=LEN(A$)
50 PRINT:PRINT ”Input number of character is ”;L;“.”
60 END
70 END PARACT
Ok
RUN
Input character string? BASIC_Unit
Input number of character is 10.
```

LET

Purpose: <u>Command or Statement.</u> Assigns the value of an expression to a variable.

Syntax: `[LET]` *variable-name* = *expression*

Comments: Evaluates *expression* and stores the result in *variable-name*.

If the types of the *variable* and *expression* cannot be converted, an error occurs.

The value can be stored in a variable that has not been declared or used before in the program, but the number of such variables is limited.

Note The `LET` keyword can be omitted completely from an assignment statement.

Example:

```
A = B + C
```

Program Sample:

```
10 ’test command name :LET
20 PARACT 0
```

```
30 LET A=5 : LET B=6 : LET C=7
40 ANS=A*B*C
50 PRINT A;″*″;B;″*″;C;″=″;ANS
60 END
70 END PARACT
Ok
RUN
5 * 6 * 7 = 210
```

LINE INPUT

Purpose: Statement. Reads an entire line into a character variable.

Syntax:
`LINE INPUT` [`WAIT` *specified time*`,`] [`″`*prompt*`″` {`,` | `;`}] *character-variable*
`LINE INPUT` *#file-no.*`,` *character-variable*

Comments: When the first form is executed, the *prompt* (if any) is displayed and the system waits for input from the terminal. If a semicolon (`;`) is specified after the *prompt*, a question mark and a space are displayed after the *prompt*. If a comma (`,`) is specified, the question mark is not displayed.

All the characters typed before the return key is pressed are stored in *character-variable*.

If `WAIT` is specified, and the input of data is not completed within the specified time, the `LINE INPUT` statement causes an error. Time is specified in units of 0.1 second; the value of the time expression must be between 1 and 32767. To wait until the input is completed, either specify a time of 0, or do not use `WAIT` at all.

The second form reads a line (up to a return code – `CHR$(13)`) from the file opened as *file-no*.

If the file specified by *file-no.* is a terminal, the effect is the same as the `LINE INPUT` statement.

Note To use the `LINE INPUT#` statement, the file must have been opened in `INPUT` mode.

Example:

```
LINE INPUT A$
```

Program Sample:

```
10 ’test command name :LINE INPUT
20 PARACT 0
30 LINE INPUT ″Input character string″;CS$
40 IF CS$=″″ GOTO 30
50 PRINT ″Character string is ″;CS$;″.″
60 END
70 END PARACT
Ok
RUN
Input character string? BASIC
Character string is BASIC.

10 ’test command name :LINE INPUT#
20 PARACT 0
30 OPEN “DATA1” FOR OUTPUT AS #1
40 FOR I=1 TO 10
50   PRINT #1, “LINE ″+STR$(I)
60 NEXT I
70 CLOSE
80 OPEN “DATA1” FOR INPUT AS #1
90 FOR I=1 TO 10
100   LINE INPUT #1,K$
```

```
110   PRINT K$
120 NEXT I
130 CLOSE
140 END
150 END PARACT
Ok
RUN
LINE 1
LINE 2
LINE 3
LINE 4
LINE 5
LINE 6
LINE 7
LINE 8
LINE 9
LINE 10

10 ’test command name :LINE INPUT WAIT
20 PARACT 0
30 LINE INPUT WAIT 100, ”Wait for input for 10 seconds”;IN$
40 PRINT ”Input character is ”;IN$;”.”
50 END
60 END PARACT
Ok
RUN
Wait for input for 10 seconds? I/O Timeout in 30
```

See Also: `INPUT`

LIST/LLIST

Purpose: Command. Displays or prints part or all of program.

Syntax:

`LIST` [*start-line-no.*] [– [*end-line-no.*]]
`LLIST` [*start-line-no.*] [– [*end-line-no.*]]

Comments: Displays the program lines from *start-line-no.* to *end-line-no.* The *start-line-no.* and *end line no.* can be line numbers that do not exist.

If only *start-line-no.* is specified, only that line is displayed.

If the *start-line-no.* is specified followed by a hyphen (–), the program lines are displayed starting from the specified line.

If *end-line-no.* is specified preceded by a hyphen (–), the program lines are displayed starting from the first line to the specified line.

If both the *start-line-no.* and *end line no.* are omitted, all the program lines are displayed.

The `LLIST` command outputs the program line(s) to the printer instead of the display. `CR + LF` is the carriage return code.

To stop the display or print, press `ABORT`.

Example:

```
LIST –999
```

LOAD

Purpose: Command. Loads BASIC program into current program area.

Syntax: `LOAD` *"file-name"*

Comments: Reads the BASIC program specified by *file-name* to the program area currently in use. The program to be read is in text format with a `.BAS` extension. `0:` can be omitted.

If *file-name* does not exist is specified, an error message is displayed.

For details on the *file-name*, refer to the description of `OPEN`.

Example: `LOAD ”0:DEMO1”`

See Also: `OPEN`, `PGEN`, `SAVE`

LOC

Purpose: Function. Gives current logical position in file.

Syntax: `LOC(`*file-no.*`)`

Comments: If the file specified by *file-no.* is a sequential file, `LOC` returns the number of bytes read or written since the file was opened.

If the specified file is a random file, `LOC` returns the number of the last record read or written.

If the file is a communication port, `LOC` returns the number of characters in the port’s input buffer.

The value returned is of double-precision floating point type.

Note If the characters in a communication port buffer are not read quickly with `INPUT#` or `INPUT$`, the buffer will fill up rapidly and an error will occur.

Example: `A = LOC(3)`

Program Sample:

```
10 ’ test command name :LOC
20 PARACT 0
30 OPEN ”DATA” AS #1
40 FIELD #1,10 AS BUF$
50 FOR I=1 TO 10
60     LSET BUF$=MKI$(I)
70     PUT #1,I
80 NEXT I
90 INPUT ”Input record number.”;A
100 IF A>10 GOTO 90
110 IF A<=0 GOTO 90
120 GET #1,A
130 PRINT ”Input record number: ”;A
140 PRINT ”Read data:           ”;CVI(BUS$)
150 PRINT ”LOC (1):             ”LOC(1)
160 CLOSE
170 END
180 END PARACT
```

LOCATE

Purpose: Statement. Moves cursor on screen.

Syntax: `LOCATE` *horizontal-position*`,` *vertical-position*

Comments: Moves the cursor on the screen to the specified *horizontal-position* and *vertical-position*.

Horizontal-position must be between 0 and 79; 0 indicates the leftmost column on the screen. *Vertical-position* must be between 0 and the number of lines per screen set with the memory switch (maximum 39).

Example: `LOCATE 10, 20`

Program Sample:

```
10 ' test command name :LOCATE
20 PARACT 0
30 CLS
40 FOR Y=1 TO 5
50   X=Y^2
60   LOCATE X,Y
70   PRINT "OMRON"
80 NEXT Y
90 END
100 END PARACT
Ok
RUN
 OMRON
    OMRON
         OMRON
               OMRON
                        OMRON
```

LOF

Purpose: Function. Returns size of file.

Syntax: `LOF (`*file-no.*`)`

Comments: If the file specified by *file-no.* is a sequential file, `LOF` returns the total size of the file.

If the specified file is a random file, `LOF` returns the number of records in the file.

If the file is a communication port, `LOF` returns the remaining number of bytes in the input buffer.

The value returned is of double-precision floating point type.

Example: `A = LOF(2)`

Program Sample:

```
10 ' test command name :LOF                    '
20 PARACT 0
25 DIM A$50
30 OPEN "DATA" AS #1
40 FIELD #1,50 AS A$
50 FOR I=1 TO 15
60    LSET A$=STRING$(50,"A")
70    PUT #1,I
80 NEXT I
90 PRINT "LOF(1)=";LOF(1)
100 END
110 END PARACT
Ok
RUN
LOF(1)=15
```

LOG

Purpose: Function. Calculates the natural logarithm of *expression*.

Syntax: `LOG (`*expression*`)`

Comments: The value of the *expression* must be greater than 0.

If the *expression* is of integer or single-precision floating point type, the logarithm is calculated and returned as a single-precision value. If *expression* is of double-precision floating point type, the logarithm is calculated and returned as a double-precision floating point number.

Example:

```
A = LOG(3)
```

Program Sample:

```
10 ' test command name :LOG                    '
20 PARACT 0
30 FOR I=1 TO 9
40   PRINT "LOG(";I;") = ";LOG(I)
50 NEXT I
60 END
70 END PARACT
Ok
RUN
LOG( 1 ) =  0
LOG( 2 ) =  .693147
LOG( 3 ) =  1.09861
LOG( 4 ) =  1.38629
LOG( 5 ) =  1.60944
LOG( 6 ) =  1.79176
LOG( 7 ) =  1.94591
LOG( 8 ) =  2.07944
LOG( 9 ) =  2.19722
```

LSET/RSET

Purpose:

Statement. To move data from memory to a random-file buffer and left- or right-justify it in preparation for a `PUT` statement.

Syntax:

`LSET` *character-variable* = *character-expression*
`RSET` *character-variable* = *character-expression*

Comments:

Places the *character-expression* into *character-variable*, which may be a field variable.

If the number of characters in *character-expression* is less than the field width, `LSET` will left-justify the characters, and `RSET` will right-justify them. Any remaining space in the field will be filled with spaces.

If the number of characters in *character-expression* is greater than the field width, the right portion of the *expression* is lost, regardless of which statement is used.

Numeric expressions can be placed into field variables by first converting them to character strings with `MKI$`, `MKS$`, or `MKD$`.

Note *Character-variable* may be a normal character variable (i.e. not one declared with the `FIELD` statement). In this case, the “field length” is simply the length of the variable’s contents.

Example:

```
LSET A$ = "ABCDEF"
```

Program Sample:

```
10 A$ = SPACE$(20)
20 RSET A$ = "HELLO"
30 PRINT A$
Ok
RUN
_______________HELLO
```

See Also:

`FIELD`, `MKI$`, `MKS$`, `MKD$`, `PUT`

MERGE

Purpose: Command. To merge lines from a BASIC program into the program already in memory.

Syntax: `MERGE "`*file-name*`"`

Comments: Reads and merges the BASIC program specified by *file-name* with the program in the current program area.

The program to be read is in text format with extension `.BAS`. The extension is not included in *file-name*.

If the program in the current program area and the specified BASIC program have any line numbers in common, lines from the merged BASIC program take precedence.

If *file-name* does not exist, an error message is displayed.

For details on the *file-name*, refer to the description of `OPEN`.

Example: `MERGE "0:PART2"`

See Also: `LOAD`, `OPEN`, `PGEN`, `SAVE`

MESSAGE

Purpose: Statement. To allocate and release message numbers.

Syntax: `MESSAGE` *function*`,` *message-no.*

Comments: *Function* 0 allocates the *message-no.*; *function* 1 releases the *message-no.*

Message no. must be between 1 and 32767.

Up to four messages in a task or eight messages in a program can be used.

If an unallocated *message-no.* is released, an error occurs.

Example: `MESSAGE 0,35`

Program Sample:

```
10 ' test command name :MESSAGE
20 PARACT 0
30 MESSAGE 0,1
40 SEND 1,"THIS IS THE MESSAGE."
50 PRINT "TASK 0: MESSAGE SENT.":TASK 1
60 END
70 END PARACT
80 '
90 PARACT 1
100 MESSAGE 0,1
110 RECEIVE 1,A$
120 PRINT "TASK 1 GOT MESSAGE: "; A$
130 END PARACT
Ok
RUN
TASK 0: MESSAGE SENT.
TASK 1 GOT MESSAGE: THIS IS THE MESSAGE.
```

See Also: `RECEIVE`, `SEND`

MID $

Purpose: Statement. To replaces part of character string variable.

Syntax: `MID$(`*character-expression-1*`,` *expression-1* `[,` *expression-2*`]) =` *character-expression-2*

Comments: Replaces characters of the *character-expression-1* after character number *expression-1* with *expression-2* characters from *character-expression-2*.

Character-expression-1 must not be a null string.

If the value of the *expression-2* is greater than the length of *character-expression-2*, only as many characters as are in *character-expression-2* are replaced.

If the *expression-2* is omitted, all the characters from the *character-expression-2* are placed in *character-expression-1* after character number *expression-1*.

The value of *expression-1* must be greater than 0 and less than the length of *character-expression-1*.

Example:

```
MID$(A$,3,7) = B$
```

Program Sample:

```
10 ’ test command name :MID$
20 PARACT 0
30 A$ = ”NUMARICAL” : PRINT A$
40 MID$(A$,4,1) = ”E”
50 PRINT A$
60 END
70 END PARACT
Ok
RUN
NUMARICAL
NUMERICAL
```

See Also: MID$ (Function)

MID $

Purpose: Function. To returns character string of specified length from specified position in character string.

Syntax: MID$(*character-expression*, *expression-1* [, *expression-2*])

Comments: Returns *expression-2* characters after character number *expression-1* from the *character-expression*.

Expression-1 must be between 1 and 538; *expression-2* must be between 0 and 538.

If the *expression-2* is omitted, or if the number of characters to the right of the character specified by *expression-1* is less than *expression-2*, all the characters to the right of *expression-1* are returned.

If *expression-1* is greater than the length of *character-expression*, a null string is returned.

Example:

```
A$ = MID$(B$,3,5)
```

Program Sample:

```
10 ’ test command name :MID$
20 PARACT 0
30 A$ = ”BASIC OMRON ASCII”
40 B$ = MID$(A$,10,7)
50 PRINT B$
60 END
70 END PARACT
Ok
RUN
ON_ASCI
```

See Also: LEFT$, RIGHT$

MKI$/MKS$/MKD$

Purpose: Function. To convert numeric value into character string.

Syntax:

`MKI$(`*integer1-value*`)`
`MKS$(`*single-precision-value*`)`
`MKD$(`*double-precision-value*`)`

Comments: `MKI$` converts an integer value into a 2-character string.

`MKS$` converts a single-precision value into a 4-character string.

`MKD$` converts a double-precision value into an 8-character string.

Note The resulting string may contain unprintable (control) characters. These functions are generally used to store numeric data in random file fields.

Example:

```
A$ = MKI$(1234%)
```

Program Sample:

```
10 ' test command name :MKI$/MKS$/MKD$
20 PARACT 0
30 DIM BUF$128
40 IA%=100%:SB=12345.6:DC#=12345.123456789#
50 OPEN "DATA" AS #1
60 FIELD #1,128 AS BUF$
70 A$=MKI$(IA%)
80 PRINT "MKI$(";IA%;") --> ";A$
90 B$=MKS$(SB)
100 PRINT "MKS$(";SB#;") --> ";B$
110 C$=MKD$(DC#)
120 PRINT "MKD$(";DC#;") --> ";C$
130 LSET BUF$=A$+B$=C$
140 PUT #1,1
150 PRINT
160 GET #1,1
170 I%=CVI(LEFT$(BUF$,2))
180 PRINT "CVI(";A$;") --> ";I%
190 S=CVS(MID$(BUF$,3,4))
200 PRINT "CVS(";B$;") --> ";S
210 D#=CVD(MID$(BUF$,7,8))
220 PRINT "CVD(";C$;") --> ";D#
230 CLOSE
240 END
250 END PARACT
Ok
RUN
MKI$( 100 ) --> d
MKS$( 12345.6 ) --> ff@F

MKD$( 12345.123456789 ) --> M H@

CVI(d) --> 100

CVS(ff@F) --> 12345.6

CVD(M H@) --> 12345.123456789
```

Note The display example shown above is when the terminal is the FIT. The displayed characters differ depending on the terminal to be used.

See Also: `CVI`, `CVS`, `CVD`

MON

Purpose: Command. Sets monitor mode.

Syntax: `MON`

Comments: Switches from BASIC mode to monitor mode.

To return to BASIC mode, press `Q` followed by return.

Example: `MON`

MSET

Purpose: Command. To set upper limit of BASIC program area (to allocate space for machine language program area).

Syntax: `MSET [`*address*`]`

Comments: The machine language program area is located before the BASIC program area. It is therefore necessary to set the upper limit of the BASIC program area to protect the machine language program area. The BASIC limit can be set between &H500 and &HFFF3.

Specify the *address* as a segment base address.

If *address* is omitted, the current upper limit is displayed.

Example: `MSET &H2000`

NAME

Purpose: Command or Statement. Changes file name.

Syntax: `NAME "`*old-file-name*`" AS "`*new-file-name*`"`

Comments: Renames *old-file-name* to *new-file-name*. Specify the complete file name, including the extension.

For details on the *file-name*, refer to the description of `OPEN`.

Note If an *old-file-name* does not exist, or if an existing file is named *new-file-name*, an error occurs. In addition, the name of an open file cannot be changed.

Example: `NAME "O:OLDNAME.BAS" AS "O:NEWNAME.BAS"`

NEW

Purpose: Command. Deletes program and variables.

Syntax: `NEW`

Comments: Deletes the program and variables in the current program area.

If a program name has been registered with the `PNAME` command, the program cannot be deleted, and a message to that effect is displayed. Delete the program name with `PNAME ""` before executing `NEW`.

Example: `NEW`

See Also: `PINF`, `PNAME`

OCT $

Purpose: Function. Returns a character string expressing the value of *expression* in octal notation.

Syntax: `OCT$(`*expression*`)`

Comments: Any fractional part of *expression* is truncated at the decimal point.

The range of the *expression* is from –32768 to 65535, and the value to returned is `"0"` to `"177777"`.

Example:

```
PRINT OCT$(321)
```

Program Sample:

```
10 ’ test command name :OCT$                     ’
20 PARACT 0
30 PRINT ”DECIMAL”, ”OCTAL”
40 FOR I=6 TO 16
50         PRINT I,OCT$(I)
60 NEXT I
70 END
80 END PARACT
Ok
RUN
DECIMAL       OCTAL
6             6
7             7
8             10
9             11
10            12
11            13
12            14
13            15
14            16
15            17
16            20
```

ON ALARM GOSUB

Purpose: Statement. Specifies interrupt time and defines interrupt routine.

Syntax: `ON ALARM` *time* `GOSUB` {*line-no.* | *label*}

Comments: Defines a processing routine to be executed when the *time* specified has elapsed.

Time is specified in units of 0.1 second in the range of 1 to 864000 (0.1 seconds to 24 hours).

The alarm interrupt is stopped immediately after the `ON ALARM GOSUB` statement has been executed.

The interrupt is enabled once for each `ON ALARM GOSUB` statement and the interrupt is stopped while the interrupt routine is being executed.

To exit from the interrupt routine, use the `RETURN` statement.

This statement can be used to define a separate alarm interrupt routine for each task.

Note

1. If more than one timer alarm is set in a task, only the last alarm set is valid.
2. The alarm interrupt will be recognized only if the `ALARM ON` statement has been executed.
3. The `ALARM OFF` state caused by the `ON ALARM GOSUB` statement continues until an `ALARM ON` statement is executed.

Example:

```
ON ALARM 1000 GOSUB *LABEL3
```

Program Sample:

```
10 ’ test command name :ALARM
20 PARACT 0
30 CLS
40 ON ALARM 300 GOSUB *WAKE.UP
60 ALARM ON
70 LOCATE 0,0:PRINT ”Set alarm timer at”;TIME$
80 PAUSE
90 END
```

```
100 *WAKE.UP
110 LOCATE 0,5 : PRINT ”Alarm at”; SPACE$(10);TIME$
120 RETURN
130 END PARACT
```

See Also: ALARM {ON | OFF | STOP}

ON COM GOSUB

Purpose: Statement. Defines routine to process communication line interrupts.

Syntax: ON COM [(*port-no.*)] GOSUB {*line-no.* | *label*}

Comments: Defines a processing routine to be executed when an interrupt occurs from a communication line specified by *port-no.*

Port-no. must be in the range of 1 to 3. If *port-no.* is omitted, 1 is assumed.

Only one interrupt routine can be defined for each *port-no.* for all tasks.

The communications interrupt is stopped immediately after the ON COM GOSUB statement is executed.

The interrupt is stopped while the interrupt routine is executed.

To exit from the interrupt routine, use the RETURN statement.

Note

1. If the ON COM GOSUB statement is executed more than once for the same *port-no.*, only the last processing routine defined is valid.
2. Communications interrupts will be recognized only if the COM ON statement has been executed.
3. The COM OFF state caused by the ON COM GOSUB statement continues until a COM ON statement is executed.

Example: ON COM(1) GOSUB 2000

Program Sample:

```
10 ’ test command name :COM ON/OFF/STOP ’
20 ’                    (TASK1)          ’
30 PARACT 0
40 PRINT ”RECEIVE PROGRAM STARTS”
50 OPEN ”COM1:N,8,2,XN” AS #1
60 ON COM (1) GOSUB *COMPRO
70 COM (1) ON
80 PRINT ”WAITS FOR RECEPTION”
90 INPUT ”RETURN: END”, IN$
100 IF IN$=”” THEN GOTO 110 ELSE 90
110 END
120 ’
130 *COMPRO
140 PRINT ”RECEIVED”
150 PRINT INPUT$(LOC(1),#1);
160 RETURN
170 END PARACT
180 ’ test command name :COM ON/OFF/STOP ’
190 ’                    (TASK2)          ’
200 PARACT 1
210  PRINT ”TRANSFER PROGRAM STARTS”
220 OPEN ”COM1:N,8,2,XN” AS #1
230 PRINT #1, ”NO ABNORMALITY UP TO NOW”
240 CLOSE :END
250 END PARACT
```

See Also: COM {ON | OFF | STOP}

ON ERROR GOTO

Purpose: Statement. Defines error processing routine and starts error trapping.

Syntax: `ON ERROR GOTO {0 | line-no. | *label}`

Comments: Defines an error processing routine to be executed when an error occurs.

If an error occurs, the `ERR` system variable contains the error number, and the `ERL` variable contains the number of the line on which the error occurred.

To exit from the error processing routine, use the `RESUME` statement.

Errors that occur while executing the error processing routine are not trapped.

To turn off the error trap, execute the `ON ERROR GOTO 0` statement.

A separate error processing routine can be defined for each task.

Example: `ON ERROR GOTO 2000`

Program Sample:

```
10 ' test command name :ON ERROR GOTO   '
20 PARACT 0
30 DIM A(10)
40 ON ERROR GOTO 90
50 FOR I=1 TO 11
60         A(I) = I
70 NEXT I
80 END
90 *SUB
100 PRINT "ERROR OCCURS."
110 RESUME NEXT
120 END PARACT
Ok
RUN
ERROR OCCURS.
```

See Also: `ERR`, `ERL`, `ERROR`, `RESUME`

ON FINS GOSUB

Purpose: Statement. Defines routine to process network interrupts.

Syntax: `ON FINS GOSUB {line-no. | *label}`

Comments: Defines a processing routine to be executed when an network input interrupt occurs.

The network interrupt is stopped immediately after the `ON FINS GOSUB` statement is executed.

The interrupt is stopped while the processing routine is being executed.

To exit from the interrupt routine and go to the previous state, use the `RETURN` statement.

Only one interrupt routine can be defined for all tasks.

Note

1. If `ON FINS GOSUB` is executed more than once, only the last processing routine defined is valid.
2. Network input interrupts will be recognized only if the `FINS ON` statement has been executed.
3. The `FINS STOP` state caused by the `ON FINS GOSUB` statement continues until a `FINS ON` statement is executed.

Example: `ON FINS GOSUB 2000`

Program Sample:

```
10 'EXECUTE THIS PROGRAM BY MACHINE NO. 2
20 PARACT 0
```

```
30 ’
40 OPEN ”FINS:00.00.26” AS #1
50 PRINT #1,”PLEASE RETURN THIS DATA”
60 INPUT #1,REVERSE$
70 PRINT ”RETURNED DATA IS ”;REVERSE$;”.”
80 CLOSE #1
90 END PARACT

10 ’EXECUTE THIS PROGRAM BY MACHINE NO. 10
20 PARACT 0
30 ’
40 OPEN ”FINS:00.00.18” AS #1
50 ON FINS GOSUB *RCV
60 FINS ON
70 PAUSE
80 CLOSE #1
90 END
100 ’
110 *RCV
120 INPUT #1,RCVD$
130 PRINT ”WHAT IS RECEIVED IS ”;RCVD$;”.”
140 PRINT #1,RCVD$
150 RETURN
160 ’
170 END PARACT
```

See Also: FINS {ON | OFF | STOP}

ON GOSUB

Purpose: Statement. Calls one of several subroutines.

Syntax: ON *expression* GOSUB {*line-no.* | *label} [, {*line-no.* | *label}]*

Comments: Branches to a subroutine having the *line-no.* or *label* corresponding to the value of the *expression*. That is, if the value of the expression is 1, the subroutine at the first *line-no.* or *label* is called; if the value is 2, the subroutine at the second *line-no.* or *label* is called, and so on.

If the value of the *expression* is 0 or if it is greater than the number of *line-nos.* or *labels*, no subroutine is called and execution continues with the next line.

If the value of the *expression* is negative, an error occurs.

To exit from a subroutine, execute the RETURN statement.

Example: ON A GOSUB 200,300,400

Program Sample:

```
10 ’test command name :ON GOSUB
20 PARACT 0
30 FOR I=1 TO 3
40          PRINT ”ON”;I;”GOSUB”
50          ON I *PRO1,*PRO2,*PRO3
60          PRINT
70 NEXT I
80 END
90 *PRO1 :PRINT ”PROGRAM1”:RETURN
100 *PRO2 :PRINT ”PROGRAM2”:RETURN
110 *PRO3 :PRINT ”PROGRAM3”:RETURN
120 END PARACT
Ok
```

```
RUN
ON 1 GOSUB
PROGRAM1

ON 2 GOSUB
PROGRAM2

ON 3 GOSUB
PROGRAM3
```

See Also: `GOSUB`, `RETURN`, `ON GOTO`

ON GOTO

Purpose: Statement. Branches to one of several specified lines.

Syntax: `ON` *expression* `GOTO` {*line-no.* | **label*} `[,` {*line-no.* | **label*}]*

Comments: Branches to a line specified by the *line-no.* or *label* corresponding to the value of *expression*. That is, if the value of the expression is 1, the program branches to the first *line-no.* or *label*; if the value is 2, the program branches to the second *line-no.* or *label*, and so on.

If the value of the *expression* is 0 or if it is greater than the number of *line-nos.* or *labels*, execution continues with the next line.

If the value of the *expression* is negative, an error occurs.

Example:

```
ON A GOTO 200,300,400
```

See Also: `GOTO`, `ON GOSUB`

ON KEY GOSUB

Purpose: Statement. Defines interrupt routine to be called when a specified numeric key is pressed.

Syntax: `ON KEY (`*key-no.*`) GOSUB` {*line-no.* | *label*}

Comments: Defines a routine to be executed when the numeric key specified by *key-no.* is pressed.

Key no. must be a number from 0 to 9.

The key interrupt is stopped immediately after the `ON KEY GOSUB` statement is executed.

The interrupt of the same key number is stopped while the interrupt routine is being executed.

To exit from the interrupt routine and return to the previous state, use the RETURN statement.

Only one interrupt routine can be defined for each *key-no.*

Note

1. If `ON KEY GOSUB` is executed more than once, only the last processing routine defined is valid.
2. Key interrupts will be recognized only if the `KEY ON` statement has been executed.
3. The `KEY OFF` state caused by the `ON KEY GOSUB` statement continues until a `KEY ON` statement is executed.

Example:

```
ON KEY (3) GOSUB 2000
```

Program Sample:

```
10 'test command name :ON KEY GOSUB                    '
20 PARACT 0
30 CLS
40 ON KEY (1) GOSUB *F1
```

```
50 KEY (1) ON
60 PRINT ”PRESS KEY 1. (END: E)”
70 PAUSE
80 END
90 *F1  : PRINT ”KEY 1 PRESSED.” : RETURN
100 END PARACT
```

See Also: `KEY` {`ON` | `OFF` | `STOP`}

ON PC GOSUB

Purpose: Statement. Defines routine to process PC interrupts.

Syntax: `ON PC (`_interrupt-no._`) GOSUB` {_line-no._ | _*label_}

Comments: This statement is used to accept an interrupt request for the data sent by the SEND instruction or received by the RECV instruction of the PC. Set the data that generates the interrupt to the bits 8 through 11 of the address indicated by the _lower CH no. of control data_ of the operand of the SEND/RECV instruction plus 1 channel in hexadecimal number (1 to F).

This statement defines a processing routine to be executed when the PC interrupt specified by _interrupt-no_. is generated. Specify a value from 1 to 15 as the _interrupt-no._

Only one interrupt routine can be specified for each _interrupt-no._

The PC interrupt is stopped immediately after the `ON PC GOSUB` statement has been executed.

The interrupt is stopped while the interrupt routine is being executed.

To exit from the interrupt routine, execute the `RETURN` statement.

Note

1. If `ON PC GOSUB` is executed more than once for the same _interrupt-no.,_ only the last processing routine defined is valid.
2. PC interrupts will be recognized only if the `PC ON` statement has been executed.
3. The `PC STOP` state caused by the `ON PC GOSUB` statement continues until a `PC ON` statement is executed.

Example: `ON PC (2) GOSUB 2000`

See Also: `PC` {`ON` | `OFF` | `STOP`}

ON SIGNAL GOSUB

Purpose: Statement. Defines interrupt routine for user-defined signal or system signal.

Syntax: `ON SIGNAL (`_signal-no._`) GOSUB` {_line-no._ | _label_}

Comments: Defines a processing routine to be executed when an interrupt is generated by the signal _signal-no._

Signal-no. must be an integer between 1 and 5 or 10 and 13. The STOP signal is 10, the PC watchdog timer error signal is 11, the cyclic error signal is 12. and the battery error signal is 13.

Signal numbers 1 through 5 are available for user definition.

The signal interrupt is stopped immediately after the `ON SIGNAL GOSUB` statement has been executed.

The interrupt is stopped while the interrupt routine is being executed.

To exit from the interrupt routine and return to the previous state, execute the `RETURN` statement.

A separate interrupt routine for user-defined signals 1 to 5 can be defined for each task. However, only one interrupt routine for each signal from 10 to 13 can be defined in all tasks.

Note

1. If `ON SIGNAL GOSUB` is executed more than once for the same *interrupt-no.*, only the last processing routine defined is valid.
2. Signal interrupts will be recognized only if the `SIGNAL ON` statement has been executed.
3. The `SIGNAL OFF` state caused by the `ON SIGNAL GOSUB` statement continues until a `SIGNAL ON` statement is executed.

Example: `ON SIGNAL 4 GOSUB 2000`

Program Sample:

```
10 'test command name :ON SIGNAL GOSUB
20 PARACT 0
30 ON SIGNAL 1 GOSUB *RCV
40 SIGNAL 1 ON
50 TASK 1
60 PAUSE
70 END
80 *RCV
90 PRINT "SIGNAL IS RECEIVED."
110 RETURN
120 END PARACT
130 '
140 PARACT 1
150 PRINT "SIGNAL IS SENT."
160 SENDSIG 1,0
170 END
180 END PARACT
Ok
RUN
SIGNAL IS SENT.
SIGNAL IS RECEIVED.
Ok
```

See Also: `SIGNAL {ON | OFF | STOP}`

ON TIME$ GOSUB

Purpose: Statement. Defines time interrupt and interrupt routine.

Syntax: `ON TIME$ = "`*time*`" GOSUB` {*line-no.* | *label*}

Comments: Defines a processing routine to be executed when the specified *time* comes.

Specify the *time* as follows:

`"HH:MM:SS"` . . . . `HH` : hour (00 to 23)
`MM` : minute (00 to 59)
`SS` : second (00 to 59)

The interrupt is stopped immediately after the `ON TIME$ GOSUB` statement has been executed.

The interrupt is stopped while the interrupt routine is being executed.

To exit from the interrupt routine and return to the previous state, execute the `RETURN` statement.

A separate interrupt routine can be defined for each task.

Note

1. If `ON TIME$ GOSUB` is executed more than once in one task, only the last processing routine defined is valid.
2. The `TIME$` interrupt will be recognized only if the `TIME$ ON` statement has been executed.
3. The `TIME$ OFF` state caused by the `ON TIME$ GOSUB` statement continues until a `TIME$ ON` statement is executed.

Example: `ON TIME$ = ”13:00:00” GOSUB 2000`

Program Sample:

```
10 ’test command name :ON TIME$ GOSUB                ’
20 PARACT 0
30 PRINT TIME$
40 INPUT ”INPUT TIME. HH:MM:SS :”;T$
50 ON TIME$ = T$ GOSUB *BEL
60 PRINT ”WAKE UP AT ”;T$;”.”
70 TIME$ ON
80 ’
90 *SLEEPY
100 PRINT ”I’M SLEEPING.”
110 PAUSE
120 ’
130 *BEL
140 PRINT
150 PRINT ”WAKE UP NOW!!”
160 TIME$ OFF
170 END
180 END PARACT
Ok
RUN
08:05:43
INPUT TIME. HH:MM:SS :? 08:10:00
WAKE UP AT 08:10:00
I’M SLEEPING.

WAKE UP NOW!!
Ok
```

See Also: `TIME$ {ON | OFF | STOP}`, `TIME$`

ON TIMER GOSUB

Purpose: Statement. Specifies time interval for recurring timer interrupt and defines interrupt routine.

Syntax: `ON TIMER =` *interval* `GOSUB` {*line-no.* | **label*}

Comments: Defines a processing routine will be executed each time the time interval specified by *interval* elapses.

Specify *interval* in units of 0.1 seconds in the range of 1 to 864000 (0.1 seconds to 24 hours).

The timer interrupt is stopped immediately after the `ON TIMER GOSUB` statement has been executed.

The interrupt is stopped while the interrupt routine is being executed. Timer processing will, however, continue.

To exit from the interrupt routine and return to the previous state, execute the `RETURN` statement.

A separate timer interrupt routine can be defined for each task.

Note

1. If `ON TIMER GOSUB` is executed more than once in any one task, only the last processing routine defined is valid.
2. Timer interrupts will be recognized only if the `TIMER ON` statement has been executed.
3. The `TIMER OFF` state caused by the `ON TIMER GOSUB` statement continues until a `TIMER ON` statement is executed.

Example: `ON TIMER = 3600 GOSUB 2000`

Program Sample:

```
10 'test command name :ON TIMER
20 PARACT 0
30 ON TIMER 50 GOSUB *DSPLY
40 TIMER ON
50 PAUSE
60 GOTO 50
70 END
80 *DSPLY
90 TIMER OFF
100 PRINT "WELCOME"
110 TIMER ON
120 RETURN
130 END PARACT
```

See Also: `TIMER {ON | OFF | STOP}`

OPEN

Purpose: Statement. Opens file.

Syntax: `OPEN` *"file-name"* `[FOR {INPUT | OUTPUT | APPEND}] AS` *#file-no.*

Comments: Opens the file specified by *file-name* as the *file-no.*

File-no. must be an integer between 1 and 15. Two files cannot be opened simultaneously using the same file number.

The following modes can be specified by `FOR`:

None: Random access file

`INPUT`: read from an existing file

`OUTPUT`: Create and output to a new file

`APPEND`: Append to the end of an existing file

In the `OUTPUT` mode, a new file is created with the specified file name. If a file with that name already exists, it is deleted.

In the `INPUT` and `APPEND` modes, an error occurs if the file specified by *file-name* does not exist.

File-name is a character string which consists of the following parts:

device name`:`[*base-name*][`.`*extension*]

The base name is an 8-character alphanumeric character string starting with an alphabetic character. The extension consists of three alphanumeric characters starting with an alphabetic character. If more than three characters are specified, an error will result.

The device names are as follows:

Name	Device
0	Memory card
COM1	Communication line port 1 (RS-232, top)
COM2	Communication line port 2 (RS-232, bottom)
COM3	Communication line port 3 (RS-422)
KYBD	Console keyboard
SCRN	Console screen
LPRT	Printer (PRT)
FINS	Network

The communication line ports are opened in the following format. Each item must be delimited from the others by a comma (`,`). (If the items are omitted, the comma is not necessary.)

OPEN "COMn	:[*speed*] [,*parity*] [,*data*] [,*stop*] [,*XON/XOFF*] [,*RS*] [,*CSm1*] [,*DS0*] [,*LF*]" AS # *file-no.*
n	Port no. Specify 1 to 3.
Speed	Integer constant indicating bit transfer rate in bit/second (bps). Specify 300, 600, 1200, 2400, 4800, 9600, or 19200. If omitted, the value defined by the memory switch (ESW4) is assumed. If the memory switch is not set, 9600 is the default value.
Parity	Transmission/reception is done with odd parity when O is specified. For even parity, specify E. N is for no parity. The default is N.
Data	Indicates bit length of a character. Specify 7 or 8. The default value is 8.
Stop	Indicates stop bit length. Specify 1 or 2. The default value is 1.
XON/XOFF (see note)	Specifies whether XON/XOFF flow control should be used. When X is specified, XON/XOFF flow control is used. When XN is specified, XON/XOFF flow control is not used. The default is to use XON/XOFF flow control. Port 3 is compatible with n:n connections. XON/XOFF flow control will thus not be used for receptions even if designated.
RS	When RS is specified, the RTS (request to send) signal is turned ON when I/O instructions are executed; otherwise, the RTS signal is turned OFF. If RS is not specified, the RTS signal is always ON. This specification is invalid when the terminal port or printer port is specified by the memory switch.
CS	Designate to monitor the time of transmission completion. The default value is CS0 (indefinite wait).
m	Indicates permissible wait time until the completion of transmission after CTS is ON. Specify this in units of 100 milliseconds from 0 to 30000. If transmission has not been completed before the wait time has elapsed, an error occurs. If the time is set to 0, the wait time is indefinite. The default value is 0.
DSO	Controls checking of DSR (data set ready) signal. If DS0 is omitted, checking is performed; otherwise, checking is not performed.
LF	Used to transfer communication file directly to printer. A LF (line-feed) character is automatically sent following every CR (carriage-return) character.

Note Port 3 corresponds to the n:n connection, so XON/XOFF flow control is not available at the time of reception.

Open the network in the following format:

OPEN "FINS: [*network-address*]*node-address*.*unit-address*" AS *#file-no.*

By OPENing the FINS device, data may be exchanged with other BASIC Units.

The network address, node address, and unit address are values specifying the address of the BASIC Unit defined for communication. Each address is specified as an absolute decimal value. Be sure to input the values correctly because the value range will not be checked thoroughly.

Network address	Distinguishes network. A routing table must be set in PC. If only one network exists, the routing table is not necessary, in which case either the network address is set to 0 or omitted.
Node address	Address of PC when network is configured using Communication Unit. If no network is used and data exchange is performed with the BASIC Unit on PC (or Expansion Base), this address must be either set to 0 or omitted.
Unit address	Address of the other BASIC Unit to/from which data is transmitted/received. The value of this address is the unit number defined by the switch on the front panel plus 16. Therefore, if the unit number is 0, the address is 16. This address is omitted if only reception is executed without the other BASIC Unit specified.

Example:

```
OPEN "FINS:1.5.20" AS #1
```

Indicates that communication is established with the BASIC Unit of machine no. 4 of network 1 and node 5.

```
OPEN "FINS:21" AS #1
```

Indicates that communication is established with BASIC Unit of machine no. 5 on the same PC.

```
OPEN "FINS:" AS #1
```

Indicates that data is received from any other BASIC Units. In this case, only input instructions such as INPUT #1 can be used.

Example:

```
OPEN "COM1:9600,E,8,1,XN" AS #1
OPEN "FINS:1.5.20" AS #1
```

Program Sample:

```
10 'test command name :ON OPEN
20 PARACT 0
30 PRINT "1:CREATE 2:MODIFY 3:LIST 9:END"
40 INPUT "INPUT NUMBER? ";IN$
50 IF IN$="9" THEN END
60 IF IN$="1" THEN OPEN "DATA" FOR OUTPUT AS #1:GOSUB *KEYIN
70 IF IN$="2" THEN OPEN "DATA" FOR APPEND AS #1:GOSUB *KEYIN
80 IF IN$="3" THEN OPEN "DATA" FOR INPUT AS #1:GOSUB *DSPLY
90 GOTO 30
100 *KEYIN
110    INPUT "PRODUCT NAME: ",NM$
120    INPUT "PRICE: ",PRC
130    PRINT #1,NM$;",";PRC
140    INPUT "INPUT END? (Y/N) ";IN1$
150    IF IN1$="Y" OR IN1$="y" THEN CLOSE:RETURN
160    IF IN1$="N" OR IN1$="n" THEN GOTO *KEYIN
170 GOTO 140
180 *DSPLY
190    PRINT "PRODUCT ANME        PRICE"
200    IF EOF(1) THEN CLOSE:RETURN
210    INPUT #1,NM$,PRC$
220    PRINT USING "    $      $    ######";NM$,VAL(PRC$)
230 GOTO 200
240 END PARACT
Ok
RUN
1:CREATE 2:MODIFY 3:LIST 9:END
INPUT NUMBER? 1
PRODUCT NAME: BASIC
PRICE: 100
INPUT END? (Y/N) Y
```

```
1:CREATE 2:MODIFY 3:LIST 9:END
INPUT NUMBER? 3
PRODUCT NAME:       PRICE:
BASIC               100
1:CREATE 2:MODIFY 3:LIST 9:END
INPUT NUMBER? 9
```

See Also: CLOSE, GET, INPUT#, PRINT#, PUT, WRITE#

OPTION BASE

Purpose: Statement. Declares subscript of first array element.

Syntax: `OPTION BASE {0 | 1}`

Comments: Declares the subscript of the first element of an array to be 0 or 1.

This declaration can be made only once, before the declaration of all variables.

If this declaration is not made, element 0 is the first element of all arrays.

Note Execute the OPTION BASE statement before the declaration or use of all variables. This makes the declaration of OPTION BASE valid in all tasks.

Example: `OPTION BASE 1`

Program Sample:

```
10 ’test command name :OPTION BASE
20 OPTION BASE 0
30 PARACT 0
40 DIM X(10)
50 FOR I=0 TO 10
60    X(I)=I*I+1
70 NEXT I
80 FOR I=0 TO 10
90    PRINT ”X(”;I;”)=”;X(I)
100 NEXT I
110 END
120 ENDPARACT
Ok
RUN
X( 0 )= 1
X( 1 )= 2
X( 2 )= 5
X( 3 )= 10
X( 4 )= 17
X( 5 )= 26
X( 6 )= 37
X( 7 )= 50
X( 8 )= 65
X( 9 )= 82
X( 10 )= 101
```

See Also: DIM, RDIM

OPTION ERASE

Purpose: Statement. Declares initialization of all non-volatile variables.

Syntax: `OPTION ERASE`

Comments: Declares that all non-volatile variables are initialized before execution.

This declaration can be made only once, before the declaration of all variables.

Note The OPTION ERASE statement must be executed before the declaration or use of all variables.

Example:

```
OPTION ERASE
```

Program Sample:

```
10 'test command name :OPTION ERASE
20 OPTION ERASE 'Operation differs depending on whether this
line is described
30 RDIM HOLD(10),HOLD2(10)
40 PARACT 0
50 HOLD(0)=HOLD(0)+1
60 HOLD2(0)=HOLD2(0)+2
70 PRINT HOLD(0)*HOLD2(0)
80 ENDPARACT
Ok
RUN
2
Ok
```

See Also: RDIM

OPTION LENGTH

Purpose: Statement. Declares default length for fixed-length character string variables.

Syntax: OPTION LENGTH *no.-of-characters*

Comments: The number of characters that can be stored in a character string variable is limited and this is called the maximum number of characters.

The maximum number of characters can be declared when a character string variable is declared. If the variable is declared or used without the maximum number of characters specified, the variable is assumed to have the maximum number of characters specified by the *no.-of-characters*.

The maximum number of characters of a character string variable is 538. Specify the *no.-of-characters* as a numeric constant in the range of 0 to 538.

This declaration can be made only once before the declaration of all variables.

If neither the declaration by the DIM or RDIM statement nor this declaration is made, the maximum number of characters is 18.

Note Execute the OPTION LENGTH statement before the declaration or use of all variables. This makes the declaration of OPTION LENGTH valid in all tasks.

Example:

```
OPTION LENGTH 32
```

Program Sample:

```
10 'test command name :OPTION LENGTH
20 OPTION LENGTH 50
30 PARACT 0
40 A$="123456789012345678 More than 18 characteres can be
displayed."
50 PRINT A$
60 END
70 END PARACT
Ok
RUN
123456789012345678 More than 18 characteres can be displayed.
```

See Also: DIM, RDIM

PARACT

Purpose: Statement. Declares the beginning of a task.

Syntax: `PARACT` *task-no.* [`WORK` *no.-of-bytes*]

Comments: This statement must be used as the first statement of a task to declare the *task-no.* and the size of the task's work area.

A task is delimited by this statement and a corresponding `END PARACT` statement.

Task-no. must be a unique integer between 0 and 15.

Task-no. 0 is the start task and is always executed first. Without this task, the program is not executed.

The *no.-of-bytes* parameter specifies the maximum amount of work memory to be used by this task. Program statements require varying amounts of memory. For example, the 4 bytes of work memory is necessary for one `GOSUB`, several bytes for one nesting of ( ) in an expression, and, in the case of a character string expression, the sum of the lengths of all the strings in the expression. If the error message `Out of memory space` is displayed, either increase *no.-of-bytes* or simplify the character string expression. If the task calls deeply nested subroutines, increase *no.-of-bytes*.

The default value for *no.-of-bytes* is 1024.

This statement cannot be included in a multi-statement.

Example: `PARACT 0`

Program Sample:

```
10 'test command name :PARACT                 '
20 PARACT 0
30 A$="ABC"
40 B$="123"
50 PRINT A$+B$
60 END
70 END PARACT
RUN
ABC123
Ok
```

See Also: `END`, `END PARACT`

PAUSE

Purpose: Statement. Stops execution of a task until an interrupt occurs.

Syntax: `PAUSE`

Comments: Stops the execution of a task until an interrupt to the task occurs.

When an interrupt occurs, program execution branches to the interrupt processing routine; when interrupt processing is completed, the task resumes execution with the statement following the `PAUSE`**.**

If an interrupt occurs for which no interrupt processing routine has been defined, the program remains stopped.

Example: `PAUSE`

Program Sample:

```
10 'test command name :PAUSE                  '
20 PARACT 0
30 ON ALARM 50 GOSUB *APRO
40 ALARM ON
50 PRINT "STARTS 5 SECONDS LATER.":PAUSE
60 INPUT "REDO? (Y OR N) "; IN$
```

```
70 IF IN$ + "Y" OR IN$ + "y" THEN 30
80 END
90 *APRO
100 PRINT "ALARM SIGNAL IS GENERATED."
110 RETURN
120 END APRACT
Ok
RUN
STARTS 5 SECONDS LATER.
ALARM SIGNAL IS GENERATED.
REDO? (Y OR N)? N
Ok
```

See Also: `ON {ALARM | COM | FINS | KEY | PC | SIGNAL | TIME$ | TIMER} GOSUB`

PC ON/OFF/STOP

Purpose: Statement. Enables, disables, or stops interrupt from PC.

Syntax: `PC (`*interrupt-no.*`) {ON | OFF | STOP}`

Comments: `ON` enables the interrupt. When this statement is executed, the program execution branches to a defined routine if an interrupt occurs.

`STOP` stops the interrupt. When this statement is executed, the program execution does not branch to a defined routine even if an interrupt occurs. However, the occurrence of the interrupt is recorded, and execution branches to the defined routine when the interrupt is enabled.

`OFF` operates the same as `STOP`.

Interrupt-no. must be an integer between 1 and 15.

Note

1. The PC interrupt is stopped immediately after the execution of the `ON PC GOSUB` statement.
2. The interrupt is stopped while the interrupt routine is being executed.

Example: `PC (1) ON`

See Also: `ON PC GOSUB`

PC READ

Purpose: Statement. Reads data from PC into *variable*.

Syntax: `PC READ [WAIT` *time*`,]` *character-expression*`;` *variable* `[,` *variable*`]*`

Comments: Reads data from a PC into *variable* according to the specifications encoded in the *character-expression* string.

If `WAIT` is specified, and if reading data is not completed before *time* has elapsed, an error occurs. Specify the *time* in units of 0.1 seconds in the range of 1 to 32767. To wait until the completion of read, either specify the *time* to be 0, or do not specify `WAIT`.

Here is the structure of the *character-expression* string:

[[*#network*`,` *node*`,`] *source-area*`,` *start-word*`,` *no.-of-words*`,`] *format* `[,` *format*]*

Network and *node* specify the address of the PC. For a PC on the same network, use *network* 0.

No network node or network address is required by a PC mounted with a BASIC Unit. If a PC address is specified, the data of the specified PC will be read.

The *source-area* parameter specifies a variable area of the PC from which data is read. If a PC address is specified with *network-address* and *node-address*, *source-area* must also be specified.

The PC address and *source-area* may be omitted to read data sent by a PC with the SEND(192) instruction. This is often used in the PC interrupt-invoked subroutine defined with `ON PC GOSUB`.

Start-word specifies the address of the desired word(s) in the PC memory; *no.-of-words* specifies the number of words to transfer. The range of *no.-of-words* is 1 to 256. If the words are consecutive, efficient operation should be expected by making a single `PC READ` statement rather than making more than one `PC READ` statement.

If the data sent from the PC is greater than the capacity of a *variable*, the data overflows and is ignored.

If the data sent from the PC is less than the capacity of a *variable*, the remaining area of the *variable* is unaffected.

`WAIT` designation with no PC address is not available.

The *format* parameter specifies how the data is to be read into the *variable*:

Source Area List

Source Area	Area	Wd/bit no.	Unit
`@R`	I/O relay, internal auxiliary relay	0 to 2555	word
`@A`	Special auxiliary relay (A) (A256 through 511 are read-only (PC READ) relays.)	0 to 511	word
`@TN`	Transition flag (read-only (PC READ))	0 to 1023	bit
`@ST`	Step flag (read-only (PC READ))	0 to 1023	bit
`@TF`	Timer flag (read-only (PC READ))	0 to 1023	bit
`@T`	Timer present value	0 to 1023	word
`@CF`	Counter flag (read-only (PC READ))	0 to 1023	bit
`@C`	Counter present value	0 to 1023	word
`@D`	Data memory (DM)	0 to 24575	word
`@E0`	Expansion DM (EM) bank 0	0 to 32765	word
⋮	⋮	⋮	⋮
`@E7`	Expansion DM (EM) bank 7	0 to 32765	word
`@SG`	CPU bus link (G) 8 words from 128 + (unit No.) x 8 channels as area for PC WRITE	0 to 255	word
`@SQ`	Cyclic output (PC A BASIC Unit PC READ)	0 to n	word
`@SI`	Cyclic input	3 to m (0 to 2 are used by system.)	word

The word/bit ranges above are for the CV1000. For other PCs, designate according to the permissible ranges. Refer to the *CV-series PC Operation Manual: Ladder Diagrams.*

Formats

For reading into simple variables:

Name	Syntax	Meaning
I	*m*I*n*	*n*-digit decimal data (*n*: 1 to 4) of *m* words
H	*m*H*n*	*n*-digit hexadecimal data (*n*: 1 to 4) of *m* words
O	*m*O*n*	*n*-digit octal data (*n*: 1 to 4) of *m* words
B	*m*B*n*	Data of *n*th bit of *m* words (*n*: 0 to 15)
A	*m*A*n*	ASCII data specified by *n* of *m* words (n: 1 to 3)

For reading into arrays:

Name	Syntax	Meaning
S	`S`*m*`I`*n* `S`*m*`H`*n* `S`*m*`O`*n* `S`*m*`B`*n*	Array data of *m* words of each type in the array type of `I`, `H`, `O`, or `B`

If *m* is omitted, 1 is assumed.

One *variable* must be supplied for each `I`, `H`, `O`, or `B`-format word.

One (string) *variable* must be supplied for each `A` in *format*.

One array variable must be supplied for each `S` in *format*. The array must be large enough to hold the data returned. Use a one-dimensional array.

Details of format

I format (*m*`I`*n*)

Each digit is treated as a decimal number (0 to 9).

Digit	Bit															
n	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	---				---				---				x 10^0			
2	---				---				x 10^1				x 10^0			
3	---				x 10^2				x 10^1				x 10^0			
4	x 10^3				x 10^2				x 10^1				x 10^0			

Example: `2I3` indicates 3-digit decimal data of two words.

H format (*m*`H`*n*)

Each digit is treated as a hexadecimal number (&H00 to &H0F).

Digit	Bit															
n	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	---				---				---				x 16^0			
2	---				---				x 16^1				x 16^0			
3	---				x 16^2				x 16^1				x 16^0			
4	x 16^3				x 16^2				x 16^1				x 16^0			

Example: `3H4` indicates 4-digit hexadecimal number of three words.

O format (*m*`O`*n*)

Each digit is treated as an octal number (&0 to &7)

Digit	Bit															
n	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	---				---				---				x 8^0			
2	---				---				x 8^1				x 8^0			
3	---				x 8^2				x 8^1				x 8^0			
4	x 8^3				x 8^2				x 8^1				x 8^0			

Example: `4O2`... indicates 2-digit octal data of four channels.

B format (*m*B*n*)

Each bit is treated as a binary number (0 or 1).

Digit	Bit															
n	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	x 2^{0}
1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	x 2^{1}	–
2	–	–	–	–	–	–	–	–	–	–	–	–	–	x 2^{2}	–	–
3	–	–	–	–	–	–	–	–	–	–	–	–	x 2^{3}	–	–	–
4	–	–	–	–	–	–	–	–	–	–	–	x 2^{4}	–	–	–	–
5	–	–	–	–	–	–	–	–	–	–	x 2^{5}	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–	x 2^{6}	–	–	–	–	–	–
7	–	–	–	–	–	–	–	–	x 2^{7}	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	x 2^{8}	–	–	–	–	–	–	–	–
9	–	–	–	–	–	–	x 2^{9}	–	–	–	–	–	–	–	–	–
10	–	–	–	–	–	x 2^{10}	–	–	–	–	–	–	–	–	–	–
11	–	–	–	–	x 2^{11}	–	–	–	–	–	–	–	–	–	–	–
12	–	–	–	x 2^{12}	–	–	–	–	–	–	–	–	–	–	–	–
13	–	–	x 2^{13}	–	–	–	–	–	–	–	–	–	–	–	–	–
14	–	x 2^{14}	–	–	–	–	–	–	–	–	–	–	–	–	–	–
15	x 2^{15}	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Example: `5B14`... indicates binary data at 14th bit position of 5 words.

A format (*m*A*n*)

Each digit is treated as an ASCII code.

Digit	Bit															
n	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	–								ASCII code							
2	ASCII code								–							
3	ASCII code								ASCII code							

Example: `6A2`... indicates ASCII code data of higher digit of 6 channels.

In format `A`, up to 256 words can be transferred at a time because one variable of the BASIC Unit correspond to more than one word of the PC.

Example:

```
PC READ ”@D,0,255,50A3,100A2,30A1,75A3”; A$, B$, C$, D$
```

`A$` receives the data specified by the `50A3` portion of *format*.
50 words * 2 characters = 100 characters are read.

`B$` receives the data specified by the `100A2` portion of *format*.
100 words * 1 character = 100 characters are read.

`C$` receives the data specified by the `30A1` portion of *format*.
30 words * 1 character = 30 characters are read.

`D$` receives the data specified by the `75A3` portion of *format*.
75 words * 2 characters = 150 characters are read.

S formats (S*m*I*n*, S*m*H*n*, S*m*O*n*, S*m*B*n*)

Each digit (bit) is treated in accordance with its specified type.

Format A must not be specified.

Format	Meanings
S*m*I*n*	Indicates *n*-digit decimal array variable data of *m* words
S*m*H*n*	Indicates *n*-digit hexadecimal array variable data of *m* words
S*m*O*n*	Indicates *n*-digit octal array variable data of *m* words
S*m*B*n*	Indicates data of n^{th} bit position of array variable data of *m* words

The *variable* corresponding to the S format must be declared as a one-dimensional array with the `DIM` statement. If the *variable* corresponding to an S format is not an array variable, an error occurs.

Data are stored sequentially in the specified array variable in accordance with the format.

In format S, one array element of the BASIC Unit must correspond to one word of the PC. Only the beginning of the array variable must be described for one format, so that up to 256 words of data can be transferred at a time.

Example:

```
PC READ ”@D,0,255,S100I4,S75H2,S80O3”;A(1),B(11),C(51)
```

`A(1)` to `A(100)`: 4-digit decimal number of 100 words indicated by `S100I4` are read.

`B(11)` to `B(85)`: 2-digit hexadecimal number of 75 words indicated by `S75H2` are read.

`C(51)` to `C(130)`: 3-digit octal number of 80 words indicated by `S80O3` are read.

Examples of PC READ Format Conversion

Source-area, *start-word*, and *no.-of-words* are omitted.

I format

PC word data

1	2	3	4

Integer type variable

```
PC READ ”I1”;J% ............ J% = 4

PC READ ”I2”;J% ............ J% = 34

PC READ ”I3”;J% ............ J% = 234

PC READ ”I4”;J% ............ J% = 1234
```

Character string variable

```
PC READ ”I1”;A$ ............ A$ = ”4”

PC READ ”I2”;A$ ............ A$ = ”34”

PC READ ”I3”;A$ ............ A$ = ”234”

PC READ ”I4”;A$ ............ A$ = ”1234”
```

H format

PC word data

8	9	A	B

Integer type variable

```
PC READ ”H1”;J% ............ J% =  &HB          = 11

PC READ ”H2”;J% ............ J% =  &HAB         = 171

PC READ ”H3”;J% ............ J% =  &H9AB        = 2475

PC READ ”H4”;J% ............ J% =  &H89AB       = –30293
```

Character string variable

```
PC READ ”H1”;A$ ............ A$ = ”B”

PC READ ”H2”;A$ ............ A$ = ”AB”
PC READ ”H3”;A$ ............. A$ = ”9AB”

PC READ ”H4”;A$ ............. A$ = ”89AB”
```

O format

PC word data

1	2	3	4

Integer type variable

```
PC READ ”O1”;J% ............. J% =  &4          = 4

PC READ ”O2”;J% ............. J% =  &34         = 28

PC READ ”O3”;J% ............. J% =  &234        = 156

PC READ ”O4”;J% ............. J% =  &1234       = 668
```

Character string variable

```
PC READ ”O1”;A$ ............. A$ = ”4”

PC READ ”O2”;A$ ............. A$ = ”34”

PC READ ”O3”;A$ ............. A$ = ”234”

PC READ ”O4”;A$ ............. A$ = ”1234”
```

B format

PC word data

C	1	2	2

Integer type variable

```
PC READ ”B1”;J% ............. J% =  2

PC READ ”B2”;J% ............. J% =  0

PC READ ”B5”;J% ............. J% =  32

PC READ ”B14”;J%  ........... J% =  16384

PC READ ”B15”;J%  ........... J% =  –32768
```

Character string variable

```
PC READ ”B1”;A$ ............. A$ = ”2”

PC READ ”B2”;A$ ............. A$ = ”0”

PC READ ”B5”;A$ ............. A$ = ”32”

PC READ ”B14”;A$  ........... A$ = ”16384”

PC READ ”B15”;A$  ........... A$ = ”–32768”
```

A format

PC word data

5	1	5	2
5	3	5	4

Error occurs if an attempt is made to read A format data into an integer variable.

Character string variable

```
PC READ ”2A1”;A$  ........... A$ = ”RT”
PC READ ”2A2”;A$ ........... A$ = ”QS”

PC READ ”2A3”;A$ ........... A$ = ”QRST”
```

Note Ascii code for: Q . . . . &H51

R . . . . &H52

S . . . . &H53

T . . . . &H54

S format

PC word data

0	1	2	3
4	5	6	7
8	9	0	1
2	3	4	5

Integer type variable (I format)

```
PC READ “S4I4”;A(1) ........ A(1) = 123
                             A(2) = 4567
                             A(3) = 8901
                             A(4) = 2345
```

Examples:

```
PC READ ”#1.5,@D,100,10,S5H4,5A3”; A%(0), B$
PC READ ”#0.3,@R,20,5,3I4,H4,A3”; A, B, C, D, E$
PC READ ”@D,3,12,10A3,S2H4”; A$, B(0)
PC READ ”2H4,S5H4”; B, C, D(0)
```

Program Sample:

```
10 ’
20 ’ WRITES DATA MEMORY OF PC, TRANSFERS DATA FROM PC, AND
COMPARES DATA READ FROM BASIC
30 ’
40 PARACT 0
50 DIM DM(3),R(3)
60 ’
70 PC WRITE ”#1.1,@D,0,3,3H4”;D1,D2,D3 ’WRITES TO DATA MEMORY
OF NETWORK 1, NODE 1
80 ON PC (1) GOSUB *RCV
90 PC (1) ON
100 PAUSE
110 PC READ ”#1.1,@D,0,3,S3H4”;DM(0) ’READS DATA MEMORY OF
NETWORK 1 WITH SPECIFIED NODE
120 FOR I= 0 TO 3
130   IF DM(I) <> R(I) THEN PRINT ”COMPARER”;I ’RECEIVE DATA
COMPARISON
140 NEXT I
150 END
160 ’
170 *RCV
180 PC READ ”S3H4”;R(0) ’READS TRANSFER DATA FROM PC
190 RETURN
200 ’
210 END PARACT
```

See Also: `PC WRITE`

PC WRITE

Purpose:

Statement. Transfers value of *variable* to PC.

Syntax:

PC WRITE [WAIT *time*,] *character-expression*; *variable* [, *variable*]*

Comments:

Writes data to a PC from *variable* according to the specifications encoded in the *character-expression* string.

If WAIT is specified, and if writing data is not completed before *time* has elapsed, an error occurs. *Time* is specified in units of 0.1 seconds, in the range of 1 to 32767. To wait until the write is complete, either specify the *time* to be 0, or do not specify WAIT.

Here is the structure of the *character-expression* string:

[[#*network*, *node*,] *destination-area*, *start-word*, *no.-of-words*,]
format [, *format*]*

Network and *node* specify the address of the PC. For a PC on the same network, use *network* 0.

The *destination-area* parameter specifies a variable area of the PC to which data is written. If a PC address is specified with *network-address* and *node-address*, *destination-area* must also be specified.

The PC address and *destination-area* may be omitted to write data to a PC which has executed the RECV(193) instruction. This is often used in the PC interrupt-invoked subroutine defined with ON PC GOSUB.

Start-word specifies the address of the desired word(s) in the PC memory; *no.-of-words* specifies the number of words to transfer. The range of *no.-of-words* is 1 to 256.

If the number of input data of the PC is greater than the *variable*, no data are stored in the remaining area.

If the number of input data of the PC is less than the *variable*, the excess *variable* is ignored.

If *variable* has a fractional part, it is truncated to an integer that does not exceeds the value of the *variable*.

If *variable* is single- or double-precision, it is converted into an integer.

If the value of *variable* exceeds &HFFFF, an error occurs.

Formats

The *format* parameter specifies how the data is to be read into the *variable*:

For writing from simple variables:

Name	Syntax	Meaning
I	*m*I*n*	*n*-digit decimal data (*n*: 1 to 4) of *m* words
H	*m*H*n*	*n*-digit hexadecimal data (*n*: 1 to 4) of *m* words
O	*m*O*n*	*n*-digit octal data (*n*: 1 to 4) of *m* words
B	*m*B*n*	Data of *n*th bit of *m* words (*n*: 0 to 15)
A	*m*A*n*	ASCII data specified by *n* of *m* words (n: 1 to 3)

For writing from arrays:

Name	Syntax	Meaning
S	S*m*I*n* S*m*H*n* S*m*O*n* S*m*B*n*	Array data of *m* words of each type in the array type of I, H, O, or B

If *m* is omitted, 1 is assumed.

One *variable* must be supplied for each I, H, O, or B-format word.

One (string) *variable* must be supplied for each A in *format*.

One array variable must be supplied for each `S` in *format*. The array must be large enough to provide the data to send. Use a one-dimensional array.

Note
1. For details on *destination-area* and *format*, refer to the description of the `PC READ` statement.
2. During the PC's CPU Bus Unit service, `PC WRITE` may write to the PC the data sent from only one Unit out of all CPU Bus Units and Communications Units. The remaining data may be written to the PC during the next and succeeding services.

Example of PC WRITE Format Conversion

Destination-area, *start-word*, and *no.-of-words* are omitted.

I format

PC word data

Integer type variable: `J%` = 1234

0	0	0	4

```
PC WRITE "I1";J%
```

0	0	3	4

```
PC WRITE "I2";J%
```

0	2	3	4

```
PC WRITE "I3";J%
```

1	2	3	4

```
PC WRITE "I4";J%
```

PC word data

Character string variable: `A$` = "1234"

0	0	0	1

```
PC WRITE "I1";A$
```

0	0	1	2

```
PC WRITE "I2";A$
```

0	1	2	3

```
PC WRITE "I3";A$
```

1	2	3	4

```
PC WRITE "I4";A$
```

H format

PC word data

Integer type variable: `J%` = –30293 = &H89AB

0	0	0	B

```
PC WRITE "H1";J%
```

0	0	A	B

```
PC WRITE "H2";J%
```

0	9	A	B

```
PC WRITE "H3";J%
```

8	9	A	B

```
PC WRITE ”H4”;J%
```

PC word data Character string variable: A$ = ”89AB”

0	0	0	8

```
PC WRITE ”H1”;A$
```

0	0	8	9

```
PC WRITE ”H2”;A$
```

0	8	9	A

```
PC WRITE ”H3”;A$
```

8	9	A	B

```
PC WRITE ”H4”;A$
```

O format

PC word data Integer type variable: J% = 668 = &1234

0	0	0	4

```
PC WRITE ”O1”;J%
```

0	0	3	4

```
PC WRITE ”O2”;J%
```

0	2	3	4

```
PC WRITE ”O3”;J%
```

1	2	3	4

```
PC WRITE ”O4”;J%
```

PC word data Character string variable: A$ = ”1234”

0	0	0	1

```
PC WRITE ”O1”;A$
```

0	0	1	2

```
PC WRITE ”O2”;A$
```

0	1	2	3

```
PC WRITE ”O3”;A$
```

1	2	3	4

```
PC WRITE ”O4”;A$
```

B format

PC word data Integer type variable: `J%` = –32749 = &H8013

0	0	0	1

```
PC WRITE ”B0”;J%
```

0	0	0	2

```
PC WRITE ”B1”;J%
```

0	0	1	0

```
PC WRITE ”B4”;J%
```

8	0	0	0

```
PC WRITE ”B15”;J%
```

An error occurs if a character string variable is supplied for a B format specification.

- Relations between B format and variable

  When B*n* is used in *format*, bit *n* of the PC is turned ON and the other bits are turned OFF if bit *n* of the variable (expressed in binary) is ON. If bit *n* of the variable is OFF, all the bits of the PC are turned OFF. In this format, only 1 bit is handled.

  Example: 1 = &H0001 = 0000 0000 0000 0001
  B0 .... ON
  B2 to B15 .... OFF

A format

An error occurs if an integer variable is supplied for an A format specification.

PC word data Character string variable: `A$` = ”QRST”

0	0	5	1
0	0	5	2

```
PC WRITE ”2A1”;A$
```

5	1	0	0
5	2	0	0

```
PC WRITE ”2A2”;A$
```

5	1	5	2
5	3	5	4

```
PC WRITE ”2A3”;A$
```

Note Ascii code for: Q . . . . &H51
R . . . . &H52
S . . . . &H53
T . . . . &H54

S format PC word data Integer type variable (I format): `A(1)` = 9876

`A(2)` = 5432

`A(3)` = 1098

`A(3)` = 7654

9	8	7	6
5	4	3	2
1	0	9	8
7	6	5	4

```
PC WRITE ”S4I4”; A(1)
```

Example:

```
PC WRITE ”#1.5,@D,100,10,S5H4,5A3”; A%(0), B$
PC WRITE ”#0.3,@R,20,5,3I4,H4,A3”; A, B, C, D, E$
PC WRITE WAIT 50,”@D,3,12,10A3,S2H4”; A$, B(0)
PC WRITE ”2H4,S5H4”; B, C, D(0)
```

Program Sample: Refer to the description of the `PC READ` statement.

See Also: `PC READ`

PEEK

Purpose: Function. Returns contents of specified memory address.

Syntax: `PEEK(`*address*`)`

Comments: Reads and returns the contents of 1 byte at *address*.

Address must be a value between &H0 and &HFFFF.

If *address* has a fractional part, it is truncated.

Address is an offset into the segment specified by the `DEF SEG` statement executed previously.

The type of the value to be returned is of integer.

Example: `PEEK(&H5000)`

Program Sample:

```
10 ’test command name :PEEK
20 PARACT 0
30 DEF SEG=&H50
40 POKE &H10,&HFF
50 PRINT PEEK(&H10)
60 END
70 END APRACT
Ok
RUN
255
Ok
```

Note Before executing this function, execute `MSET &H1000`.

See Also: `DEF SEG`, `POKE`

PGEN

Purpose: Command. Selects current program area.

Syntax: `PGEN` [*program-no.*]

Comments: Specify an integer from 1 to 3 as *program-no.*

The program area specified by the memory switch is assumed on start.

If *program-no.* is omitted, information on the current program area is displayed in the format of the `PINF` command.

The information on the program area can also be checked by the `PINF` command.

Note

1. The program area is divided into three portions.
2. If the memory is protected, the `PGEN` command is not effective.

Example: `PGEN 2`

See Also: `PINF`

PINF

Purpose: Command. Displays information on program area.

Syntax: `PINF`

Example: `PINF`

Comments: Displays the size of a program name and S-CODE area. An asterisk (`*`) is displayed to the left of the current program area, with the size of the E-CODE area and variable area. Finally, the size of S-CODE area and that of unused memory area are displayed.

The display format is as follows:

```
NO. PNAME  S–CODE  E–CODE  GLOBAL  LOCAL
–––+––––––+––––––-+–––––––+––––––-+––––-
* 1 TEST        41      20       4     4
  2              4
  3              4
FREE         64207  110935
```

S-CODE	Area storing user BASIC program and machine language program
E-CODE	Area storing execution code of user program
Variable area	Area storing global variable and local variable
Memory area	E-CODE area + variable area

Note

1. `*` indicates the current program area. The size of “E-CODE”, “GLOBAL”, and “LOCAL” of the area are displayed.
2. The name shown in the “PNAME” column is the name given by the `PNAME` command.
3. The “S-CODE” is 4 if the program does not exist.

See Also: `NEW`, `PGEN`, `PNAME`

PNAME

Purpose: Command. Registers or deletes name of program area currently used.

Syntax: `PNAME` *"program-name"*

Comments: The *program-name* is an alphanumeric character string of 8 characters long. The uppercase characters and lowercase characters are distinguished.

If a null string (`""`) is specified as the *program-name*, the program name is deleted.

Named program areas are protected against the `NEW` instruction.

Note

1. It is not possible to register the same name for two program areas.
2. If an attempt is made to set already existing *program-name*, a message is output.

Example: PNAME ”TEST37”

See Also: PGEN, PINF, NEW

POKE

Purpose: Statement. Writes data to specified address of memory.

Syntax: POKE *address*, *expression*

Comments: Writes the value of *expression* to the byte at offset *address* in the segment set by a previous DEF SEG statement.

Address is a 2-byte integer in the range of &H0 to &HFFFFF.

Expression is a 1-byte integer in the range of &H0 to &HFF.

Caution *Address* should be within memory allocated by the MSET instruction. Poking values into other memory areas can cause erratic operation.

Example: POKE &H4500, &H29

Program Sample:

```
10 ’test command name :POKE
20 ’ MSET &H1000 HAS TO BE EXECUTED.
30 PARACT 0
40 ’
50 DEF SEG=&H500
60 FOR ADR=0 TO 40
70   READ MAC:POKE ADR,MAC
80 NEXT ADR
90 ’
100 FIRST%=12
150 DEF SEG=&H500
160 CULUCU%=&H0
170 CALL CULUCU%(FIRST%, SECOND%)
180 PRINT SECOND%
190 ’
200 END
210 ’machine language routine
220 DATA XX, XX, XX, XX, ...
230 DATA XX, XX, XX, ...
240 DATA XX, XX, ...
250 END PARACT
```

See Also: DEF SEG, MSET, PEEK

PRINT/LPRINT

Purpose: Command or Statement. Displays/prints value of an expression. CR+LF is the carriage return code.

Syntax:

{PRINT | ?} [*expression*] [{, | ; | _} [*expression*]]*

PRINT *#file-no.*, [*expression*] [{, | ; | _} [*expression*]]*

LPRINT [*expression*] [{,|;|_} [*expression*]]*

Comments: The PRINT statement displays the value of the *expression*. If more than one *expression* is given, each must be separated from the others by a comma (,), semicolon (;), or blank (_).

If the *expression* is delimited by a comma and is of 12 characters or less, the value of the expression is output starting from the beginning of the next field (each field is 14 characters wide). If the expression is of more than 12 characters, the value is output starting from the next field (in this case, each field is 28 characters wide). If the expression is delimited by a semicolon or blank, the value of the expression is output following the output result immediately before.

If there is no semicolon or comma at the end of the list of *expression*s, a carriage return is performed after all the expressions have been output.

If a numeric expression is used, one blank is necessary before and after the expression, and the blank before the expression is treated as a negative sign.

The `?` statement is the same as the `PRINT` statement.

The `PRINT #` statement is output to a file specified by the *file-no.*

The `LPRINT` statement prints the value of a specified expression. `CR+LF` is the carriage return code.

Note To use the `PRINT #` statement, open the file in the OUTPUT mode or APPEND mode.

Example:

```
PRINT ”DATA =”; A, C3
```

Program Sample:

```
10 ’test command name :PRINT
20 PARACT 0
30 X=70:Y=90
40 PRINT ”X = ”;X,”B = ”;Y
50 PRINT
60 PRINT ”X + Y =”;X+Y
70 END
80 END PARACT
Ok
RUN
X = 70       B = 90
X + Y = 160
```

```
10 ’test command name :PRINT#
20 PARACT 0
30 OPEN ”DATA1” FOR OUTPUT AS #1
40 PRINT #1,”BASIC UNIT”
50 PRINT #1,”CV500”
60 CLOSE
70 ’
80 ’Reads data from sequential file and outputs it.”
90 OPEN ”DATA1” FOR INPUT AS #1
100 IF EOF(1) GOTO 140
110 INPUT #1,A$
120 PRINT A$
130 GOTO 100
140 CLOSE
150 END
160 END PARACT
Ok
RUN
BASIC UNIT
CV500
```

See Also: `WRITE`

PRINT USING/LPRINT USING

Purpose: Command or Statement. Output value of expression in specified format.

Syntax:

`PRINT USING` *format* `;` *expression* [{`,` | `;` | _} [*expression*]]*
`PRINT` *#file-no.*`,` `USING` *format* `;` *expression* [{`,` | `;` | _} [*expression*]]*
`LPRINT USING` *format* `;` *expression* [{`,` | `;` | _} [*expression*]]*

Comments: The `PRINT USING` statement edits the value of the *expression* in accordance with the *format* and displays the value.

Characters other than format control characters (described below) in the *format* are output as-is.

If the number of digits of the *format* of the specified numeric value is less than the number of digits of the *expression*, a percent mark is (%) is prefixed to the output numeric value. If the number of digits of the output numeric value is greater than the number of digits of the *format*, a percent mark is also prefixed.

The `PRINT #` statement outputs the value to a file specified by *file-no.*

The `LPRINT USING` statement outputs the value to the printer.

The following format control characters may be used in the *format* string:

Symbol	Meanings
`!`	Displays only the first one character (1 byte) of a given character string.
`&&`	Encloses n blanks between the ampersands: `"&___&"`. Displays (n+2) bytes from the beginning of a given character string. If the character string is shorter than (n+2), it is left-justified with the rest of the character positions filled with blank.
`@`	Displays a given character string as-is.
`#`	Constitutes a numeric area with several "#". The total number of digits (including sign) is specified by the number of "#". If the data to be output falls short of the specified number of digits, the value is right-justified with the rest of the character positions filled with blank. If the value exceeds the number of digits, "%" is prefixed to the numeric value. See Note 1.
`.`	Specifies the position of the decimal point in the numeric value area. The numeric value below the decimal point is truncated according to the number of # specified on the right for display. If the value is a fraction and falls short of the number of digits, 0 is filled.
`+`	Specifies the position of the sign of the numeric value when specified at the beginning or end of the numeric area.
–	If suffixed to the numeric area, – is output after the numeric value if the numeric value is negative. If this character is prefixed or two of it are specified in a row, they are treated in the same manner as non-control characters.
`**`	Fills the left of the numeric area with asterisks (`*`) when specified at the beginning of a numeric field. The "`*`" allocates room for one digit.
`\\`	Prefixes "\" to the numeric value when specified at the beginning of a numeric field. The "\" allocates room for one digit. See Note 2.
`**\`	Combines the effects the two previous control characters when specified at the beginning of a numeric field. See Note 2.
`,`	Separates each three digits of the integer portion of a number with "`,`" when specified at any position of the numeric field. The "`,`" allocates room for one digit.
`^^^^`	When specified at the end of a numeric field, causes the value to be output in exponential format.
`_`	Outputs 1 character in the following format character string as is.

Note When using the `PRINT # USING` statement, open the file in the OUTPUT mode or APPEND mode.

Example:

```
PRINT USING "####.###"; A
```

Program Sample:

```
10 'test command name :PRINT USING
20 PARACT 0
```

```
30 A$="BASIC":B=1234.56:C=-123
40 PRINT USING "&                   &";A$
50 PRINT USING "CHARACTER STRING @ ";A$
60 PRINT USING "#####.###";B
70 PRINT USING "+####.###";B
80 PRINT USING "+####.###";C
90 PRINT USING "**#####.##";B
100 PRINT USING "\\######.##";B
110 PRINT USING "**\#####.##";B
120 PRINT USING "##,###.###";B
130 PRINT USING "#.###^^^^";B
140 PRINT USING "B=###.##";B
150 END
160 END PARACT
Ok
RUN
BASIC
CHARACTER STRING BASIC
 1234.560
+1234.560
 -123.000
***1234.56
   ¥1234.56
***¥1234.56
 1,234.560
1.235E+03
B=%1234.56
```

```
10 'test command name :PRINT# USING
20 PARACT 0
30 DIM A$50
40 OPEN "DATA1" FOR OUTPUT AS #1
50 A$="OMRON BASIC"
60 PRINT #1, USING "&                          &";A$
70 PRINT #1, USING "&       &";A$
80 CLOSE
90 '
100 OPEN "DATA1" FOR INPUT AS #1
110 IF EOF(1) GOTO 150
120 LINE INPUT #1,A$
130 PRINT A$
140 GOTO 110
150 CLOSE
160 END
170 END PARACT
Ok
RUN
OMRON BASIC
OMRON
```

See Also: PRINT

PUT

Purpose: Statement. Writes data to random file.

Syntax: PUT *#file-no.* [, *record-no.*]

Comments: Writes data to the random file specified by *file-no.*

If the *record-no.* is omitted, the record following that specified by the last PUT or GET statement is assumed. If neither statement has been executed, 0 is assumed.

The minimum value of the *record-no.* is 1; the maximum value is 32767.

Note

1. An error occurs if this statement is executed on a file opened in the sequential mode.
2. Each record is 256 bytes long.

Example: PUT #1,17

Program Sample:

```
10 ’test command name :PUT#
20 PARACT 0
30 DIM BUF$20
40 OPEN ”DATA” AS #1
50 FIELD #1,20 AS BUF$
60 FOR I=1 TO 5
70           READ RE$
80          LSET BUF$=RE$
90          PUT #1,I
100 NEXT I
110 ’
120 INPUT ”INPUT RECORD NO. (1 TO 5): ”,A
130 IF A>5 GOTO 120
140 IF A<1 GOTO 120
150 GET #1, A
160 PRINT ”RECORD NO.: ”;A, BUF$
170 CLOSE
180 END
190 ’
200 DATA ”omron”,”OMRON”,”CV500”,”BASIC”,”ABCD”
210 END PARACT
Ok
RUN
INPUT RECORD NO. (1 TO 5): 2
RECORD NO.: 2          OMRON
```

See Also: GET, FIELD, OPEN

RANDOMIZE

Purpose: Statement. Sets the seed for the random series.

Syntax: RANDOMIZE [*expression*]

Comments: Changes the random series obtained by the RND function by giving a new seed value for the random number generator.

If the *expression* is omitted, a message prompting for input is displayed.

If the input is requested, specify a seed for the random number in the range of –32768 to 32767.

The initial value of the random number seed is 0.

Example: RANDOMIZE I

Program Sample:

```
10 ’test command name :RANDOM
20 PARACT 0
30 GOSUB *RNDSUB
40 RANDOMIZE INT(RND(1)*10 + 1)
50 PRINT: PRINT ”CHANGES TO NEW RANDOM SERIES.”
60 GOSUB *RNDSUB
70 END
80 ’
90 *RNDSUB
100 FOR I=1 TO 10
110   PRINT INT (RND(1)*10) + 1;
120 NEXT I
130 PRINT
140 RETURN
150 END PARACT
Ok
RUN
3  1  4  10  10  9  1  5  9  5

CHANGES TO NEW RANDOM SERIES.
6  5  6  1  1  9  1  8  5  7
```

See Also: RND

READ

Purpose: Statement. Reads data from DATA statement and stores it in *variable*.

Syntax: READ *variable* [, *variable*]*

Comments: Reads data sequentially from the DATA statement(s) in the task and stores them in *variable*.

The first READ statement in a task takes data from the DATA statement with the lowest line number in the task; subsequent READs take data from subsequent DATA statements. One READ can take data from more than one DATA statement, and one DATA statement can supply data for more than one READ.

The RESTORE statement can be used to set the DATA position from which the next READ statement will obtain its data.

Numeric constants in DATA statements will be read as character strings if *variable* is a character string variable.

Character constants in DATA statements can also be read as a numeric values if *variable* is a numeric type. However, if the character constant cannot be converted into a numeric variable, an error occurs.

If the data defined by the task’s DATA statements has run out when a READ statement is executed, an error occurs. In addition, executing READ in a task that has no DATA statements causes an error.

Note Other task’s DATA statements cannot be read.

Example:

```
READ A,C,E
```

Program Sample:

```
10 ’test command name :READ
20 PARACT 0
30 CLS
40 FOR I=1 TO 12
50   READ A$
60   IF A$=”” THEN PRINT
70   PRINT A$;
```

```
80 NEXT I
90 END
100 DATA ”0”,”M”,”R”,”O”,”N”,””,”B”,”A”,”S”,”I”,”C”,””
110 END PARACT
Ok
RUN
OMRON
BASIC
```

See Also: `DATA`, `RESTORE`

RECEIVE

Purpose: Statement. Receives message.

Syntax: `RECEIVE` *message-no.*`,` *character-variable*

Comments: Receives a message specified by *message-no.* and stores it in *character-variable*.

If the specified *message-no.* doesn’t exist, the system waits until a message with that number is sent from another task with the `SEND` statement.

Message-no. must have been allocated by the `MESSAGE` statement. If an unallocated *message-no.* is specified, an error occurs.

Example: `RECEIVE 3,A$`

See Also: `MESSAGE`, `SEND`

REM

Purpose: Statement. Allows comments to be included in program.

Syntax: `REM [`*comment-text*`]`

Example: `REM **** SAMPLE PROGRAM ****`

Comments: Any text between `REM` and the end of the line is treated as a comment that has no influence on the program execution.

A single quotation mark (`’`) can be used instead of `REM`.

Note The comment continues until the end of the line; colons (`:`) appearing in *comment-text* are ignored.

Program Sample:

```
10 ’test command name :REM
20 PARACT 0
30 REM COMMENT
40 ’COMMENT
50 PRINT ”Statement following REM is a comment.”
60 REM PRINT ”This statement will not be executed.”
70 END
80 END PARACT
Ok
RUN
Statement following REM is a comment.
```

RENUM

Purpose: Command. Re-numbers program lines.

Syntax: `RENUM` [*new-line-no.*] [`,` [*old-line-no.*] [`,` *increment*]]

Example: `RENUM 1000,10`

Comments: Changes the program line numbers starting from the line specified by *old-line-no.* into those starting from *new-line-no.* Subsequent line numbers increase by *increment*.

If *old-line-no.* is omitted, re-numbering starts from the first line.

If both *new-line-no.* and *increment* are omitted, both are assumed to be 10.

If the specification is made in a manner such that the line numbers would no longer be in ascending order as a result of re-numbering, an error occurs.

Line numbers referenced by `GOTO` and `GOSUB` instructions are also changed.

RESTORE

Purpose: Statement. Specifies the `DATA` statement to be read by subsequent `READ` statements.

Syntax: `RESTORE` [{*line-no.* | **label*}]

Comments: Subsequent `READ` statements will read data from the `DATA` statement at the specified *line-no.* or **label.*

If *line-no.* or *label* is omitted, the first `DATA` statement in the task is selected.

Note Other task’s `DATA` statements cannot be specified for reading.

Example: `RESTORE 700`

Program Sample:

```
10 ’test command name :RESTORE
20 PARACT 0
30 DIM A$(10),B(10)
40 FLG=0
50 FOR I=1 TO 3
60         READ A$(I),B(I)
70 NEXT I
80 FOR I=1 TO 3
90         PRINT A$(I),B(I)
100 NEXT I
110 PRINT
120 IF FLG=0 THEN RESTORE *STARTDATA :FLG=1:GOTO 50
130 END
140 ’
150 DATA ASCII,1
160 DATA UNIT,2
170 *STARTDATA
180 DATA OMRON,3
190 DATA BASIC,4
200 DATA UNIT,5
210 END PARACT
Ok
RUN
ASCII        1
UNIT         2
OMRON        3

OMRON        3
BASIC        4
UNIT         5
```

See Also: `READ`, `DATA`

RESUME

Purpose: Statement. Exits from error processing routine.

Syntax: `RESUME` [{`0` | *line-no.* | **label* | `NEXT`}]

Comments: Exits from the error processing routine defined by the `ON ERROR GOTO` statement.

The `RESUME` statement is executed in the error processing routine.

If *line-no.* or **label* is omitted, or if 0 is specified, re-executes the statement in which the error occurred.

If `NEXT` is specified, resumes execution at the statement following the one where the error occurred.

If *line-no.* or *label* is specified, execution resumes at the specified line.

Example:

```
RESUME NEXT
```

Program Sample:

```
10 ’test command name :RESUME                 ’
20 PARACT 0
30 DIM BOX$(3)
40 BOX$(1)=”CODE1”
50 BOX$(2)=”CODE2”
60 BOX$(3)=”CODE3”
70 ON ERROR GOTO *ERRTRN
80 *KEYIN
90 INPUT “INPUT NUMERAL. (END:999)”; N
100 IF N=999 GOTO 130
110 PRINT ”THE CODE OF PRODUCT NO. ”;N;” IS ”;BOX$(N);”.”
120 GOTO *KEYIN
130 END
140 *ERRTRN
150 IF ERR=9 THEN PRINT ”PRODUCT NO. ”;N;” DOES NOT EXIST.”
160 PRINT
170 RESUME *KEYIN
180 END PARACT
Ok
RUN
INPUT NUMERAL. (END:999)? 1
THE CODE OF PRODUCT NO. 1 IS CODE 1.
INPUT NUMERAL. (END:999)? 4
PRODUCT NO. 4 DOES NOT EXIST.

INPUT NUMERAL. (END:999)? 999
```

See Also: `ERR/ERL`, `ERROR`, `ON ERROR GOTO`

RIGHT$

Purpose: Function. Returns character string of the specified length from right of *character-expression*.

Syntax: `RIGHT$(`*character-expression*`,` *expression*`)`

Comments: Returns a character string of length specified by *expression* from the right of *character-expression*.

If *expression* is 0, returns a null string.

If the value of *expression* exceeds the number of characters in *character-expression*, returns the entire *character-expression*.

Example:

```
B$ = RIGHT$(A$,3)
```

Program Sample:

```
10 ’test command name :RIGHT$
20 PARACT 0
30 ALL$=”BASIC UNIT”
40 PART$=RIGHT$(ALL$,4)
50 PRINT PART$
60 END
70 END PARACT
Ok
```

```
RUN
UNIT
```

See Also: LEFT$, MID$

RND

Purpose: Function. Returns a random number.

Syntax: RND(*expression*)

Comments: Returns a random number between 0 and 1.

The random number that is generated depends on the value of *expression* as follows:

If *expression* is negative, the random series is initialized.

If *expression* is 0, the previously generated random number is returned.

If *expression* is positive, the next random number in the series is returned.

The random series can be changed by the RANDOMIZE statement.

Note *Expression* must be a numeric value between –32768 and 32767*.*

Example:

```
RD = RND(1)
```

Program Sample:

```
10 'test command name :RND                    '
20 PARACT 0
30 RANDOMIZE
40 PRINT "GENERATES RANDOM NUMBER IN THE RANGE OF 1 TO 100."
50 FOR I=1 TO 10
60     N=INT(RND(1)*100+1)
70     PRINT N
80 NEXT I
90 END
100 END PARACT
Ok
RUN
Random number seed (-32768 to 32767)? 3
GENERATES RANDOM NUMBER IN THE RANGE OF 1 TO 100.
 22
 28
 72
 72
 38
 87
 53
 8
 49
 61
```

See Also: RANDOMIZE

ROMLOAD

Purpose: Command. Reads information from EEPROM to user program area.

Syntax: ROMLOAD

Comments: Reads all the information existing in the EEPROM to the user program area.

Note

1. The user program area consists of three areas defined by the PGEN command and a machine language program area.

2. The EEPROM is a ROM area for storing the user program. If the EEPROM is not installed, this command causes an error.

Example: `ROMLOAD`

See Also: `PGEN`, `ROMSAVE`, `ROMVERIFY`

ROMSAVE

Purpose: Command. Writes information from user program area to EEPROM.

Syntax: `ROMSAVE`

Comments: Writes all the information in the BASIC program, machine language program, and user program source code areas to the EEPROM.

Note

1. The user program area consists of three areas defined by the `PGEN` command and a machine language program area.
2. The EEPROM is a ROM area for storing the user program. If the EEPROM is not installed, this command causes an error.

Example: `ROMSAVE`

See Also: `PGEN`, `ROMLOAD`, `ROMVERIFY`

ROMVERIFY

Purpose: Command. Verifies between EEPROM and user program area.

Syntax: `ROMVERIFY`

Comments: Verifies the contents of the current user program area with the contents of the EEPROM.

If the contents of the current user program area do not match the contents of the EEPROM, the message “Verify Error” is displayed.

Note

1. The user program area consists of three areas defined by the `PGEN` command and a machine language program area.
2. The EEPROM is a ROM area for storing the user program. If the EEPROM is not installed, this command causes an error.

Example: `ROMVERIFY`

See Also: `PGEN`, `ROMLOAD`, `ROMSAVE`

RUN

Purpose: Command or Statement. Starts program execution.

Syntax: `RUN ["`*file-name*`"] [, ERASE]`

Comments: If *file-name* is not specified, and if the program area and contents of the program previously executed have not been changed, the current execution code is executed immediately. In this case, the execution code and changes in the program are backed up by a battery even if the power is turned off in the middle of the execution.

The program is automatically re-compiled on the following occasions:

When a new program is created
When the program is executed for the first time after it has been loaded
When the program area or contents have been changed
If a file name is specified

It takes about 18 seconds to compile 100 lines of BASIC code.

Before the execution of the program, all the variables except the non-volatile variables are initialized and any open files are closed.

If ERASE is specified, the contents of the non-volatile variables are initialized.

If *file-name* is specified, the LOAD instruction is executed before the RUN command. The file named *file-name* should be in text format, and the should have the extension .BAS. The extension should not be included in *file-name*.

The execution of the program is terminated when all the tasks have been stopped by END, STOP, or EXIT statements, or if the abort key is pressed.

A name currently registered with the PNAME instruction cannot be specified as the *file-name*.

For details on *file-name*, refer to the description of OPEN.

Example:

```
RUN
RUN "0:TEST2"
```

See Also: OPEN, PINF

SAVE

Purpose: Command. Saves BASIC program to a file.

Syntax: SAVE "*file-name*"

Comments: Saves the BASIC program in the current program area in a specified file.

The program is saved as a text-format file with extension .BAS. The extension should not specified in *file-name*.

If a file with the same name already exists, the old file overwritten by the new one.

For details on *file-name*, refer to the description of OPEN.

0: can be omitted.

Note
1. 0: at the beginning of a file name means the memory card of the PC. To use a memory card as a file device, format it with a tool (such as a FIT10) connected to the PC.
2. When attempting to save a file in a memory card that has no available bytes, only the filename will be saved.

Example:

```
SAVE "0:FILE8"
```

See Also: LOAD, PGEN, OPEN

SEARCH

Purpose: Function. Searches for a specified integer value from elements of array variable and returns the element number.

Syntax: SEARCH(*integer-array* [, *expression*] [, *start-element*] [, *increment*])

Comments: Searches for the value specified by *expression* in the elements of the array specified by *integer-array* and returns the element number found first.

If *expression* has a fractional part, it is truncated.

If the specified value is not found in *integer-array*, –1 is returned.

Integer-array must be a one-dimensional array.

Start-element specifies the position of the array from which the search is to be started. The minimum allowable value for this parameter depends on the subscript for the first element set with the OPTION BASE statement. If *start-element* is omitted, the search is started from the beginning of the array.

Increment specifies units in which the element number to be searched is incremented. If the *increment* is omitted, 1 is assumed.

Example:

```
NUMB = SEARCH(IND%,100,0,5)
```

Program Sample:

```
10 ’test command name :SEARCH
20 PARACT 0
30 DIM IND%(500)
40 FOR I= 1 TO 500  ’Fill an array with random numbers
50       IND%(I)=INT(RND(1)*100)+1
60 NEXT I
70 *KEYIN
80       SUCC=0:COUNT=0
90       PRINT ”SEARCHES RATE OF OCCURRENCE OF RANDOM NUMBER.”
100      INPUT ”INPUT NUMERALS 1 TO 100 (END:0)”;EN
110      IF EN=0 THEN END
120      IF EN<0 OR EN>100 THEN GOTO *KEYIN
130      SUCC=SEARCH(IND%,EN,SUCC+1)
140      IF SUCC<>-1 THEN COUNT=COUNT+1:GOTO 130
150   RES=COUNT/5:PRINT
160      PRINT USING ”THE RATE OF OCCURENCE OF #### IS
####.##%.”;EN;RES
170      PRINT
180 GOTO *KEYIN
190 END PARACT
Ok
RUN
SEARCHES RATE OF OCCURRENCE OF RANDOM NUMBER.
INPUT NUMERALS 1 TO 100 (END:0)? 37

THE RATE OF OCCURENCE OF   37 IS    1.20%.

SEARCHES RATE OF OCCURRENCE OF RANDOM NUMBER.
INPUT NUMERALS 1 TO 100 (END:0)? 98

THE RATE OF OCCURENCE OF   98 IS    1.80%.

SEARCHES RATE OF OCCURRENCE OF RANDOM NUMBER.
INPUT NUMERALS 1 TO 100 (END:0)? 0
Ok
```

SEND

Purpose: Statement. Sends a message.

Syntax: `SEND` *message-no.*`,` *character-expression*

Comments: Sends *character-expression* as message number *message-no.*

A character string expression of up to 538 bytes can be sent.

Message-no. must have been allocated with the `MESSAGE` statement. If *message-no.* has not been allocated, an error occurs.

Example: `SEND 4,B$`

See Also: `MESSAGE`, `RECEIVE`

SENDSIG

Purpose: Statement. Generates a signal.

Syntax: `SENDSIG` *signal-no.*`,` *task-no.*

Comments: Sends signal *signal-no.* to the task specified by *task-no.*

Signal-no. must be an integer from 1 to 5 or 10 to 13. Signals 1 to 5 are available for user definition; signals 10 to 13 have pre-defined meanings.

(Signal 10 is STOP, 11 is PC watchdog timer error, 12 is cyclic error, and 13 is battery error.)

If a *task-number* that does not exist in the program is specified, an error occurs.

Example:

```
SENDSIG 5,1
```

See Also: `ON SIGNAL GOSUB`, `SIGNAL ON/OFF/STOP`

SGN

Purpose: Function. Returns the sign of *expression*.

Syntax: `SGN(`*expression*`)`

Comments: If the value of *expression* is positive, `SGN` returns 1. If the value is 0, `SGN` returns 0. If the value is negative, `SGN` returns –1.

Expression may be an integer, single-precision, or double-precision expression.

Example:

```
PRINT SGN(A)
```

Program Sample:

```
10 ’test command name :SGN                    ’
20 PARACT 0
30 PRINT ”INPUT INTEGER”;
40 INPUT X
50 SIGN = SGN(X)
60 PRINT ”SGN(”;X;”) = ”;SIGN
70 END
80 END PARACT
Ok
RUN
INPUT INTEGER? 15
SGN( 15 ) = 1
```

SIGNAL ON/OFF/STOP

Purpose: Statement. Enables, disables, or stops signal interrupt.

Syntax: `SIGNAL` *signal-no.* {`ON` | `OFF` | `STOP`}

Comments: The *signal-no.* is an integer from 1 to 5 or 10 to 13. Signals 1 to 5 are available for user definition; signals 10 to 13 have pre-defined meanings. (Signal 10 is STOP, 11 is PC watchdog timer error, 12 is cyclic error, and 13 is battery error.)

`ON` enables the interrupt. When this statement is executed, the program execution will branch to a defined processing routine if an interrupt occurs.

`OFF` disables the interrupt. When this statement is executed, the program execution will not branch to a defined processing routine even if an interrupt occurs.

`STOP` stops the interrupt. When this statement is executed, the program execution does not branch to a defined processing routine. However, the occurrence of the interrupt is recorded, and the execution will branch to the defined routine when the interrupt is later enabled.

Note

1. The interrupt is stopped immediately after the `ON SIGNAL GOSUB` statement is executed.
2. The interrupt is stopped while the interrupt processing routine is being executed.

Example:

```
SIGNAL 5 ON
```

See Also: `ON SIGNAL GOSUB`

SIN

Purpose: Function. Calculates the sine of *expression*.

Syntax: `SIN (`*expression*`)`

Comments: Returns the sine of the specified *expression*.

Specify the *expression* in radians.

If *expression* is of integer or single-precision type, the result is a single-precision value. If it is of double-precision floating-point type, the result is a double-precision value.

Example:

```
PRINT SIN(1)
```

Program Sample:

```
10 ’test command name :SIN                     ’
20 PARACT 0
30 PRINT ”INPUT ANGLE.”
40 INPUT X
50 S=SIN(3.14159/180*X)
60 PRINT ”SIN(3.14159/180*”;X:”) = ”;S
70 END
80 END PARACT
Ok
RUN
INPUT ANGLE.
? 30
SIN(3.14159/180*30) = .5
```

See Also: `COS, TAN`

SPACE$

Purpose: Function. Returns a character string containing a specified number of spaces (_).

Syntax: `SPACE$(`*expression*`)`

Comments: The range of *expression* is from 0 to 538.

If the *expression* is 0, a null string is returned.

Note The `SPC` function is used with the `PRINT` or `LPRINT` statement to output blanks, while the `SPACE$` function returns a character string containing spaces (_).

Example:

```
A$ = SPACE$(N) + ”***”
```

Program Sample:

```
10 ’test command name :SPACE$
20 PARACT 0
30 DIM WRT$20
40 ’RIGHT-JUSTIFIED OUTPUT
50 FOR X%=0% TO 9%
60     VALUE=INT (5%^X%)
70     STRG$=STR$ (VALUE)
80     COLUMN=LEN (STRG$)
90     WRT$=”5 TO THE ”+STR$(X%)+” POWER”+SPACE$(12–COLUMN)
+STRG$
100    PRINT WRT$
110 NEXT X%
120 END
130 END PARACT
Ok
```

```
RUN
5 TO THE 0 POWER               1
5 TO THE 1 POWER               5
5 TO THE 2 POWER              25
5 TO THE 3 POWER             125
5 TO THE 4 POWER             625
5 TO THE 5 POWER            3125
5 TO THE 6 POWER           15625
5 TO THE 7 POWER           78125
5 TO THE 8 POWER          390625
5 TO THE 9 POWER     1.95313E+06
```

See Also: SPC, STRING

SPC

Purpose: Function. Outputs blanks.

Syntax: SPC(*expression*)

Comments: SPC is used in PRINT or LPRINT statements to output a specified number of blanks.

Expression must be in the range of –32768 to 32767.

If *expression* is negative, 0 is assumed.

If *expression* is larger than the width of the screen (80), it is divided by the screen width and the remainder is used as the number of blanks to output.

If *expression* has a fractional part, it is truncated.

Note

1. The SPC function cannot be used alone. It must be used in a PRINT or LPRINT statement.
2. The SPC function is used with a PRINT or LPRINT statement to output blanks, while the SPACE$ function returns blank as a character string.

Example: PRINT SPC(8); A$

Program Sample:

```
10 'test command name :SPC
20 PARACT 0
30 FOR I=1 TO 4
40          PRINT "OMRON";SPC(I);"BASIC"
50 NEXT I
60 END
70 END PARACT
Ok
RUN
OMRON_BASIC
OMRON__BASIC
OMRON___BASIC
OMRON____BASIC
```

See Also: SPACE$, TAB

SQR

Purpose: Function. Calculates the square root of *expression*.

Syntax: SQR(*expression*)

Comments: The value of *expression* must be greater than or equal to 0.

If the value of *expression* is of integer or single-precision type, the result is a single-precision value. If it is of double-precision type, then the result is a double-precision value.

Example: `C = SQR(3)`

Program Sample:

```
10 ’test command name :SQR
20 PARACT 0
30 PRINT ”    N    SQR(N)     SQR(N)^2”:PRINT
40 FOR I=1 TO 10
50 PRINT USING ”### ##.##### ###.#####”;I;SQR(I),SQR(I)^2
60  NEXT I
70   END
80 END PARACT
Ok
RUN
      N      SQR(N)      SQR(N)^2
      1      1.00000      1.00000
      2      1.41421      2.00000
      3      1.73205      3.00000
      4      2.00000      4.00000
      5      2.23607      5.00000
      6      2.44949      6.00000
      7      2.64575      7.00000
      8      2.82843      8.00000
      9      3.00000      9.00000
     10      3.16228     10.00000
```

STEP

Purpose: <u>Command.</u> Executes program one step at a time.

Syntax: `STEP`

Comments: Executes one statement of the task after being interrupted by `BREAK` or `STOP`.

`Quit in` *line-no* will be displayed when the program is stopped with CTRL + X, in which case `STEP` or `CONT` cannot be used. If `BREAK in` *line-no* is displayed when the program is stopped with CTRL + C, it is possible to use `STEP` or `CONT`.

Example: `STEP`

STOP

Purpose: <u>Statement.</u> Stops program execution.

Syntax: `STOP`

Comments: Stops the execution of all the program’s tasks.

When this statement has been executed, the following message is displayed:

`Stop in` *nnn* (*nnn* is a line number)

To resume the program execution, execute the `CONT` instruction.

Example: `STOP`

Program Sample:

```
10 ’test command name :STOP
20 PARACT 0
30 FOR I=1 TO 100
40           IF 10-I=0 THEN STOP
50           PRINT I
60 NEXT I
70 END
80 END PARACT
Ok
```

```
RUN
 1
 2
 3
 4
 5
 6
 7
 8
 9
STOP IN 40
```

See Also: CONT

STR$

Purpose: Function. Converts *expression* into character string.

Syntax: STR$(*expression*)

Example: A$ = STR$(135)

Comments: Returns a character string expressing the value of *expression*.

Expression may be integer, single-precision, or double-precision.

If the value of *expression* is positive, a blank is prefixed to the character string returned. If it is negative, a negative sign (–) is prefixed.

See Also: VAL

Program Sample:

```
10 ’test command name :STR$                   ’
20 PARACT 0
30 ’NUMERAL VALUE AND CHARACTER COMPARISON
40 FOR N=1 TO 5
50           ADD = 1+N
60           N$=MID$(STR$(N),2,1)
70           ADD$=”1”+N$
80           PRINT ”1 + ”;N;” = ”;ADD
90           PRINT ”1 + ”;N$;” = ”;ADD$
100          PRINT
110 NEXT N
120 END
130 END PARACT
Ok
RUN
1 + 1 =  2
1 + 1 = 11

1 + 2 =  3
1 + 2 = 12

1 + 3 =  4
1 + 3 = 13

1 + 4 =  5
1 + 4 = 14

1 + 5 =  6
1 + 5 = 15
```

STRING$

Purpose: Function. Returns a character string containing a specified number of copies of a character.

Syntax: STRING$(*expression*, {*character-string* | *character-code*})

Comments: Returns a character string containing *expression* copies of the first character of *character-expression* or *character-code*.

Character-code, if specified, must be from 0 to 538. The value of *expression* must be from 0 to 538.

If *expression* is 0, a null string is returned.

See Also: SPACE$

Example: A$ = STRING$(12,"$")

Program Sample:

```
10 'test command name :STRING$                 '
20 PARACT 0
25 DIM MARK$256
30 CLS
40 FOR I=1 TO 8
50           J=I^2
60     MARK$=STRING$(J,"0")
70     PRINT MARK$
80 NEXT I
90 FOR K=8 TO 1 STEP -1
100          L=K^2
110          MARK$=STRING$(L,"0")
120                 PRINT MARK$
130 NEXT K
140 END
150 END PARACT
Ok
RUN
0
0000
000000000
0000000000000000
0000000000000000000000000
000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000
000000000000000000000000000000000000
0000000000000000000000000
0000000000000000
000000000
0000
0
```

SWAP

Purpose: Statement. Swaps the values of two variables.

Syntax: SWAP *variable-name*, *variable-name*

Comments: The variables to be swapped must be of the same type.

Example: SWAP XY,PQ

Program Sample:

```
10 ’test command name :SWAP
20 PARACT 0
30 PRINT ”EXECUTES SWAP (X,Y) STATEMENT.”
40 INPUT ”X =”;X
50 INPUT ”Y =”;Y
60 PRINT ”X =”;X, ”Y =”;Y :PRINT
70 SWAP X,Y
80 PRINT ”SWAP(X,Y)”:PRINT
90 PRINT ”X =”;X ”Y =”;Y
100 END
110 END PARACT
Ok
RUN
EXECUTES SWAP (X,Y) STATEMENT.
X = ? 13
Y = ? 16
X = 13        Y = 16

SWAP(X,Y)

X = 16        Y = 13
```

TAB

Purpose: Function. Moves cursor to specified position.

Syntax: `TAB(`*expression*`)`

Comments: Moves the cursor to the column specified by *expression*.

If the specified number of columns is less than the displayed data, a carriage return will be executed.

The value of *expression* must be in the range of –32768 to 32767.

If *expression* is negative, 0 is assumed.

If *expression* is larger than the width of the screen (80), it is divided by 80 and the remainder is used as the column to move to.

If the *expression* has a fractional part, it is truncated.

Note The `TAB` function cannot be used alone; it must be used with a `PRINT` or `LPRINT` statement.

Example: `PRINT TAB(12);A$`

Program Sample:

```
10 ’test command name :TAB
20 PARACT 0
30 FOR I=15 TO 25
40            PRINT ”OMRON”
50            PRINT TAB(I);”Corporation”
60 NEXT I
70 END
80 END PARACT
Ok
RUN
OMRON_______Corporation
OMRON________Corporation
OMRON_________Corporation
OMRON__________Corporation
OMRON___________Corporation
OMRON____________Corporation
OMRON_____________Corporation
```

```
OMRON_____________Corporation
OMRON______________Corporation
OMRON_______________Corporation
OMRON________________Corporation
```

See Also: SPC

TAN

Purpose: <u>Function.</u> Calculates tangent of *expression*.

Syntax: TAN(*expression*)

Comments: Returns the tangent of the specified *expression*.

Specify the *expression* in radians.

If the *expression* is of integer or single-precision type, the result is a single-precision value. If it is of double-precision type, the result is a double-precision value.

Example: A = TAN(123)

Program Sample:

```
10 ’test command name :TAN                     ’
20 PARACT 0
30 INPUT ”INPUT ANGLE”;B
40 PI=3.14159
50 C=COS(PI/180*B)
60 T=TAN(PI/180*B)
70 S=C*T
80 PRINT ”TAN(PI/180*”;B;”) = ”;T
90 PRINT ”COS(PI/180*”;B;”) = ”;C
100 PRINT ”SIN(PI/180*”;B;”) = ”;S
110 END
120 END PARACT
Ok
RUN
INPUT ANGLE? 45
TAN(PI/180*45) = 0.999999
COS(PI/180*45) = 0.707107
SIN(PI/180*45) = 0.707106
```

See Also: COS, SIN

TASK

Purpose: <u>Statement.</u> Starts a task.

Syntax: TASK *task-no.*

Comments: Starts a task specified by *task-no.*

Task-no. can be an expression.

The execution of the task is started from the PARACT statement.

If the specified task is executing (or paused, waiting for an interrupt), an error occurs.

If *task-no.* does not exist in the program, an error occurs.

Example: TASK 7

See Also: EXIT, END PARACT, PARACT, STOP

TIME $

Purpose: Function. Returns time of internal clock, or sets time.

Syntax: `TIME$ [="hour:minute:second"]`

Comments: If "= "*hour*:*minute*:*second*"" is omitted, `TIME$` returns the current time as a character string.

If "= "*hour*:*minute*:*second*"" is specified, sets the specified time.

The format of the time is as follows:

"HH:MM:SS" HH: hour (00 to 23)
MM: minute (00 to 59)
SS: second (00 to 59)

Note If the date (`DATE$`) is abnormal, `TIME$` cannot be set correctly. When using the PC for the first time, or if the date is abnormal, set the correct date (using `DATE$`) before setting the time.

Example:

```
PRINT TIME$
```

Program Sample:

```
10 'test command name :TIME$                  '
20 PARACT 0
30 T$=TIME$
40 HH$=LEFT$(T$,2)
50 MM$=MID$(T$,4,2)
60 SS$=RIGHT$(T$,2)
70 PRINT "PRESENT TIME IS ":HH$;":";MM$;":";SS$;"."
80 END
90 END PARACT
Ok
RUN
PRESENT TIME IS 11:41:14.
```

See Also: `DATE$`

TIME$ ON/OFF/STOP

Purpose: Statement. Enables, disables, or stops time interrupt.

Syntax: `TIME$ {ON | OFF | STOP}`

Comments: `ON` enables the interrupt. When this statement is executed, the program execution will branch to a defined processing routine if an interrupt occurs.

`OFF` disables the interrupt. When this statement is executed, the program execution will not branch to a defined processing routine even if an interrupt occurs.

`STOP` stops the interrupt. When this statement is executed, the program execution will not branch to a defined processing routine if an interrupt occurs. However, the occurrence of the interrupt is recorded, and the execution will branch to the defined routine when the interrupt is later enabled.

Note

1. The interrupt is stopped immediately after the `ON TIME$ GOSUB` statement has been executed.
2. The interrupt is stopped while the interrupt processing routine is being executed.

Example:

```
TIME$ ON
```

See Also: `ON TIME$ GOSUB`

TIMER ON/OFF/STOP

Purpose: Statement. Enables, disables, or stops timer interrupt.

Syntax: `TIMER {ON | OFF | STOP}`

Comments: `ON` enables the interrupt. When this statement is executed, the program execution will branch to a defined processing routine if an interrupt occurs.

`OFF` disables the interrupt. When this statement is executed, the program execution will not branch to a defined processing routine even if an interrupt occurs.

`STOP` stops the interrupt. When this statement is executed, the program execution will not branch to a defined processing routine if an interrupt occurs. However, the occurrence of the interrupt is recorded, and the execution branches to the defined routine when the interrupt is later enabled.

Note

1. The interrupt is stopped immediately after the `ON TIMER GOSUB` statement has been executed.
2. The interrupt is stopped while the interrupt processing routine is being executed.

Example: `TIMER ON`

See Also: `ON TIMER GOSUB`

TROFF

Purpose: Command or Statement. Stops output of line number trace.

Syntax: `TROFF` [{*task-no.* | `ALL`}]

Comments: If the *task-no.* is specified, line number tracing of the specified task is stopped.

Task-no. can be an expression.

If `ALL` is specified, line number tracing of all tasks is stopped.

If *task-no.* and `ALL` are omitted, the current task is assumed.

This statement can be used in a program as a BASIC statement.

Example: `TROFF`

See Also: `TRON`

TRON

Purpose: Command or Statement. Starts output of line number trace.

Syntax: `TRON` [{*task-no.* | `ALL`}]

Comments: If the *task-no.* is specified, line number tracing of the specified task is started. If `TRON` is input by changing the line number, the line number will be added. In order not to trace the previous trace talk, use `TROFF`.

Task-no. can be an expression.

If `ALL` is specified, line number tracing of all the tasks is started.

If *task-no.* and `ALL` are omitted, the current task is assumed.

If a *task-no.* that does not exist in the program is specified, a message to that effect is output.

This statement can be used in a program as a BASIC statement.

Example: `TRON`

See Also: `TROFF`

TWAIT

Purpose: Statement. Waits for termination of task.

Syntax: `TWAIT` *task-no.*

Comments: Waits until the task specified by the *task-no.* terminates.

If an interrupt for which a processing routine is defined occurs, the waiting task execution branches to the processing routine and terminates the `TWAIT` statement.

If a *task-no.* that does not exist in the program is specified, an error occurs.

Example: `TWAIT 3`

Program Sample:

```
10 'test command name :TWAIT
20 PARACT 0
30 PRINT "WAITS UNTIL TASK 1 TERMINATES."
40 TASK 1
50 TWAIT 1
60 PRINT "TASK 1 TERMINATED."
70 END
80 END PARACT
90 '
100 PARACT 1
110 FOR I=0 TO 10
120 PRINT "TASK 1 PROCESSING..."
130 NEXT I
140 END
150 END PARACT
Ok
RUN
WAITS UNTIL TASK 1 TEMINATES.
TASK 1 PROCESSING...
TASK 1 PROCESSING...
TASK 1 PROCESSING...
TASK 1 PROCESSING...
TASK 1 PROCESSING...
TASK 1 PROCESSING...
TASK 1 PROCESSING...
TASK 1 PROCESSING...
TASK 1 PROCESSING...
TASK 1 PROCESSING...
TASK 1 PROCESSING...
TASK 1 TERMINATED.
```

See Also: `TASK`, `EXIT`

USR

Purpose: Function. Calls machine language function in memory.

Syntax: `USR[`*func-no.*`](`*argument*`)`

Comments: Calls a machine language function defined by the `DEF USR` statement.

Func-no. is a number between 0 and 9. *Func-no.* must correspond to the number defined by the `DEF USR` statement. If *func-no.* is omitted, 0 is assumed.

The *argument* passes a value to the machine language function.

The type of the value returned is the same as the type of the *argument*.

Note Before executing the `USR` function, the segment address must be made the same as that defined by the `USR` function by using the `DEF SEG` statement.

Example:

```
N = USR2(A)
```

See Also: CALL, DEF USR

VAL

Purpose: Function. Converts digits in *character-expression* into a numeric value.

Syntax: VAL(*character-expression*)

Comments: Returns the numeric value expressed by *character-expression.* Three symbols, plus (+), minus (–), and ampersand (&) can appear at the front of the *character-expression* to indicate positive, negative, or octal values. The octal and hexadecimal indicators, &O and &H, can also appear at the beginning of the *character-expression*. (If & or &O appears at the beginning of *character-expression*, the digits 8 and 9 cannot be included in the number. If &H appears at the beginning of *character-expression*, the letters A to F can be included as digits in the number.)

An exponent can be specified at the end of a decimal number, using E or D.

Type declarators %, !, and # are also valid.

If a character that cannot be converted to a numeric value is included in the *character-expression*, only the characters preceding that character are converted into a numeric value; the subsequent characters are ignored.

If *character-expression* cannot be converted to a numeric value at all, 0 is returned.

Any spaces in *character-expression* are ignored.

Example:

```
A = VAL(”7”)
```

Program Sample:

```
10 ’test command name :VAL                    ’
20 PARACT 0
30 A$=”1000”
40 PRINT ”STRING”,A$
50 PRINT ”VALUE”,VAL(A$)
60 PRINT ”OCTAL”,VAL(”&0”+A$)
70 END
80 END PARACT
Ok
RUN
STRING       1000
VALUE        1000
OCTAL        512
```

See Also: STR$

VARPTR

Purpose: Function. Returns *variable-name*’s address in memory.

Syntax: VARPTR(*variable-name*) [, *feature*]

Comments: Gives the memory address in the variable area where the data of the variable named *variable-name* is stored.

Feature, if specified, must be 0 or 1. When 0 is specified, the offset address is returned. If it is 1, the segment address is returned. If *feature* is omitted, 0 is assumed.

The type of the value returned is integer.

The location indicated by the value returned by VARPTR is as follows:

Type	Meaning
Integer	Integer type consists of 2 bytes: bytes 0 and 1. Byte 0 has the lower byte of data, while byte 1 has the higher byte of the data. The value returned by VARPTR indicates byte 0.
Single-precision	Single-precision numbers consist of 4 bytes: bytes 0 through 3. Data is expressed in the IEEE 32-bit single-precision floating-point number format. The value returned by VARPTR indicates byte 0.
Double-precision	Double-precision floating-point number type consists of 8 bytes: bytes 0 through 7. Data is expressed in the IEEE 64-bit double-precision floating-point number format. The value returned by VARPTR indicates byte 0.
Character	Character variables have the maximum length of the character string in the first 2 bytes (bytes 0 and 1) and the actual length of the string in the second 2 bytes (bytes 2 and 3). The character string data is stored starting at the 4th byte. The memory required to store a character string is the maximum length of the string + two bytes to store the maximum length + two bytes to store the actual length. If the number of bytes required to store the string is odd, an extra pad byte is added at the end of the string storage area. The content of the pad byte is undefined. The value returned VARPRT indicates byte 0.
Array	The configuration of an array variable is dependent on the type of each element and the number of dimensions in the array. The value returned by VARPTR is address of the first element of the array.

For the storage format of each type, refer to *Appendix A: Memory Storage Formats of Variables.*

Example:

```
PRINT HEX$(VARPTR(A,0))
```

Program Sample:

```
10 'test command name :VARPTR                        '
20 PARACT 0
30 A%=0
40 B=0
50 C#=0
60 D$="DUMY"
70 VARA$=HEX$(VARPTR(A%))
80 VARB$=HEX$(VARPTR(B))
90 VARC$=HEX$(VARPTR(C#),1)
100 VARD$=HEX$(VARPTR(D$),1)
110 PRINT "OFFSET ADDRESS OF VARIABLE A% IS ";VARA$;"."
120 PRINT "OFFSET ADDRESS OF VARIABLE B IS " ;VARB$;"."
130 PRINT "SEGMENT ADDRESS OF VARIABLE C# IS ";VARC$;"."
140 PRINT "SEGMENT ADDRESS OF VARIABLE D$ IS ":VARD$;"."
150 END
160 END PARACT
Ok
RUN
OFFSET ADDRESS OF VARIABLE A% IS &H4.
OFFSET ADDRESS OF VARIABLE B IS &H6.
SEGMENT ADDRESS OF VARIABLE C# IS &H1810.
SEGMENT ADDRESS OF VARIABLE D$ IS &H1811.
```

VERIFY

Purpose: Command. Verifies program.

Syntax: `VERIFY "file-name"`

Comments: Verifies the contents of the current program area with the contents of the specified file.

The file must be in text format, and the name must have the extension `.BAS`. The extension is not included in *file-name*.

If the contents of the current program area do not coincide with those of the specified file, the message `"Verify Error"` is displayed.

If *file-name* does not exist is specified, a message to that effect is output.

For the details on the *file-name*, refer to the description of `OPEN`.

`0:` can be omitted.

Example:

```
VERIFY "0:FILEA3"
```

See Also: `OPEN`

VLOAD

Purpose: Command or Statement. Reads contents of non-volatile variable area from a file.

Syntax: `VLOAD "file-name"`

Comments: Reads the contents of specified file into the non-volatile variable area.

When the memory card is used, a binary file with the extension `.BRD` is read. The extension is not included in *file-name*.

If *file-name* does not exist, a message to that effect is output.

This command can also be used in a program as a BASIC statement.

Note If the layout of the non-volatile variable area has been changed since the binary file was saved with `VSAVE`, the non-volatile variables may not have the expected values.

Example:

```
VLOAD "0:FILEB2"
```

See Also: `VSAVE`, `OPEN`

VSAVE

Purpose: Command or Statement. Saves contents of the non-volatile variable area to a file.

Syntax: `VSAVE "file-name"`

Comments: Saves the contents of the non-volatile variable area to the file specified by *file-name*.

The file is saved in binary format with the extension `.BRD`. The extension is not included in *file-name*.

For the details on *file-name*, refer to the description of `OPEN`.

This command can also be used in a program as a BASIC statement.

Note `0:` at the beginning of the file name indicates the memory card of the PC. To use the memory card as a file device, format it in advance with a tool (such as the FIT10) connected to the PC.

Example:

```
VSAVE "0:ABC1"
```

See Also: VLOAD, OPEN

WHILE/WEND

Purpose: Statement. Repeatedly execute a series of statements while *conditional-expression* is true.

Syntax:

```
WHILE conditional-expression
WEND
```

Comments: Repeatedly executes the statements between WHILE and WEND while the value of the *conditional-expression* is true (not equal to zero).

If *conditional-expression* becomes false (zero), the repetition ends, and the statements following WEND are executed.

If *conditional-expression* is false (0) from the start, the statements between WHILE and WEND are never executed, and the execution goes on to the statements following WEND.

The WHILE and WEND statements must be used in pairs.

WHILE/WEND statements may be nested. An error will result if the number of nestings exceeds 100.

Example:

```
WHILE A = 0
WEND
```

Program Sample:

```
10 ’test command name :WHILE WEND
20 PARACT 0
30 I=1
40 WHILE I<10
50           PRINT I;”–”;
60           J=1
70           WHILE J<I
80                  PRINT J;:J=J+1
90           WEND
100           PRINT :I=I+1
110 WEND
120 END
130 END PARACT
Ok
RUN
 1 –
 2 – 1
 3 – 1  2
 4 – 1  2  3
 5 – 1  2  3  4
 6 – 1  2  3  4  5
 7 – 1  2  3  4  5  6
 8 – 1  2  3  4  5  6  7
 9 – 1  2  3  4  5  6  7  8
```

See Also: FOR TO STEP/NEXT

WRITE

Purpose: Command or Statement. Outputs the value of expression.

Syntax:

WRITE *expression* [{, | ; | _} [*expression*]]*
WRITE *#file-no.*, *expression* [{, | ; | _} [*expression*]]*

Comments: The WRITE statement displays the result of calculation of the *expression* like the PRINT statement.

If more than one *expression* is specified, each must be delimited from the others by a comma (,), semicolon (;), or blank (_). (The delimiter can be any of these characters, and the expressions are output in the same manner regardless of which delimiter is used.)

The differences between this statement and `PRINT` statement are that, with `WRITE`, the unnecessary blanks are deleted, each output expression is delimited from the others by a comma (`,`), and character strings are enclosed in a pair of double quotation marks (`””`).

If there is no semicolon or comma after the last *expression*, all the expressions are output and then a carriage return is performed.

If the last *expression* ends with a semicolon or comma, an error occurs.

The `WRITE #` statement outputs the value to the file specified by *file-no.*

Note To use the `WRITE #` statement, the file must be opened in the OUTPUT or APPEND mode.

Example:

```
WRITE A
```

Program Sample:

```
10 ’test command name :WRITE
20 PARACT 0
30 D1=123:D2=456:D3$=”789”:D4$=”OMRON”
40 WRITE ”WRITE”,D1,D2,D3$,D4$
50 PRINT
60 PRINT ”PRINT ”,D1,D2,D3$,D4$
70 END
80 END PARACT
Ok
RUN
“WRITE”,123,456,“789”,“OMRON”

PRINT        123          456          789          OMRON

10 ’test command name :WRITE#
20 PARACT 0
30 OPEN ”DATA1” FOR OUTPUT AS #1
40 FOR I=1 TO 10
50           WRITE #1,I*I
60 NEXT I
70 CLOSE
80 OPEN ”DATA1” FOR INPUT AS #1
90 IF EOF(1) GOTO 130
100 INPUT #1,I$
110 PRINT I$
120 GOTO 90
130 CLOSE
140 END
150 END PARACT
Ok
RUN
1
4
9
16
25
36
49
64
81
100
```

See Also: `PRINT`

3-3 GP-IB Instructions

CMD DELIM

Purpose: Statement. Specifies the delimiter to use for `PRINT@`, `INPUT@`, and `LINE INPUT@`.

Syntax: `CMD DELIM =` *delimiter-code*

Comments: Here are the delimiter codes and the corresponding delimiters:

Delimiter code	Delimiter
0	CR + LF
1	CR
2	LF
3	EOI

EOI is always taken as a delimiter, regardless of the current delimiter setting. `CR+LF` is the default setting of the delimiter.

Example: `CMD DELIM = 1`

CMD PPR

Purpose: Statement. Selects PPR (Parallel Poll Response) mode.

Syntax: `CMD PPR =` *mode*

Comments: Specifies a response (PPR) mode for the parallel poll of the GP-IB controller.

Mode	PPR mode
0	Do not respond to parallel poll (Returns inverted data for the Master's response request)
1	Respond to parallel poll (Returns data for the Master's response request)
2	Respond to parallel poll when SRQ is transferred (Returns data for the Master's response request)

Refer to the `PPOLL` for the pattern of data.

This statement is used in slave mode.

The default value of this mode is 0.

Example: `CMD PPR = 2`

CMD TIMEOUT

Purpose: Statement. Specifies limit value for timeout check.

Syntax: `CMD TIMEOUT =` *timeout-parameter*

Comments: Specifies a time limit in the range of 0 to 255 when the statement of the GP-IB is executed. An illegal function call error will result if a value outside the range is designated, in which case the limit value previously set will not be changed.

Timeout-parameter specifies the time in seconds. When 0 is specified, the timeout check is not performed.

The default value is 0.

Example: `CMD TIMEOUT = 10`

INPUT @

Purpose: Statement. Receives data sent from specified talker and stores it in *variable*.

Syntax: `INPUT@` [*talker-address* `[,` *listener-address* `[,` *listener-address*]*]]`;` *variable* `[,` *variable*]*

Comments: In the master mode, ATN is made true and the UNL command, talker address, listener address, and MLA (my listen address) are sequentially transmitted. After that, ATN is made false and data transmitted from a specified talker is received and stored in the specified *variable*.

In the slave mode, the talker address and listener address are not specified. The system waits until the listener is addressed by the controller. When ATN is made false and data is transmitted from the talker after MLA has been received, the data is received and stored in the specified *variable*.

To more than one variable, each data is substituted each time “`,`” is received.

The type of the transmitter side and that of the receiver side must be matched.

If a character variable is specified, any spaces before and after the received value are ignored. If the data exceeds the size of a character variable, it is substituted to the next character variable.

An IEEE Timeout error will result if data reception does not complete in time when `CMD TIMEOUT` is designated. The time value range is 0 to 255 s with 1-second increments. Specify zeroes to wait until the completion of data reception.

Example:

```
INPUT@1,2;A$,B$
```

IRESET REN

Purpose: Statement. Makes REN (remote enable) false.

Syntax: `IRESET REN`

Comments: Transmits a “false” REN message (makes the REN line “H”). This statement is used in the master mode.

Example:

```
IRESET REN
```

ISET IFC

Purpose: Statement. Transmits IFC (interface clear).

Syntax: `ISET IFC [,` *integer*]

Example:

```
ISET IFC,5
```

Comments: After transmitting a “true” IFC message, makes the message false again.

The range of the *integer* is from 1 to 255. An illegal function call error will result if a value outside the range is designated, in which case the limit value previously set will not be changed.

The IFC message transmission time is dependent on *integer* as follows:

integer	Transmission Time
1 to 5	*integer* * 100ms
6 to 100	10 ms
101 to 200	20 ms
201 to 255	30 ms

If *integer* is omitted, 1 is assumed.

This statement is used in the master mode.

ISET REN

Purpose: Statement. Makes REN (remote enable) true.

Syntax: `ISET REN`

Comments: Transmits a “true” REN message (makes the REN line “L”).

This statement is used in the master mode.

When this statement is executed, the devices in the system can be remote-controlled. The devices in the system are set in the remote state when later addressed as listeners.

Almost all the GP-IB devices can be controlled from the front panel or through the GP-IB bus. The status in which the devices can be controlled from the front panel is the local status, and the status in which they can be controlled through the GP-IB bus (i.e., the status in which the devices cannot be controlled from the front panel) is the remote status.

Even in the remote status, the devices are set in the local status provided they are not locally locked out when the LOCAL key on the front panel is pressed.

By locally locking out the devices, the LOCAL switch on the front panel of a GP-IB device is locked, so as to prevent the operator of the device from pressing the switch by mistake and the system from malfunctioning.

To locally lock out the device, the `LLO` (local lock out) command must be transmitted.

Example: `WBYTE &H11;`

When the device is locally locked out, execute the `IRESET REN` statement to restore the device to the local status.

Example: `ISET REN`

ISET SRQ

Purpose: Statement. Transmits SRQ (service request).

Syntax: `ISET SRQ [@] [N]`

Comments: Transmits a “true” SRQ (service request) (makes the SRQ line “L”).

If `N` is omitted, waits until serially polled by the controller. When MTA (my talk address) is sent from the controller as a result of the serial polling, and when ATN becomes false, makes SRQ false, and then transmits the device status stored in variable STATUS to the controller.

When `@` is specified, EOI becomes true as soon as the device status has been transmitted.

This statement is used in the slave mode.

Example: `ISET SRQ@`

LINE INPUT@

Purpose: Statement. Receives string data sent from specified talker and stores it in a character string variable.

Syntax: LINE INPUT@ [*talker-address* [, *listener-address* [, *listener-address*]*]]; *character-string-variable*

Comments: In the master mode, ATN is made true, and the UNL command, talker address, listener address, and MLA (my listen address) are sequentially transmitted. After that, ATN is made false, and data transmitted from a specified talker is received and stored in *character-string-variable*, until a delimiter is received.

In the slave mode, the talker address and listener address are not specified. The system waits until the listener is addressed by the controller. When ATN is made false and data is transmitted from the talker after MLA has been received, the data is received and stored in *character-string-variable* until a delimiter is received.

An IEEE Timeout error will result if data reception does not complete in time when `CMD TIMEOUT` is designated. The time value range is 0 to 255 s with 1-second increments. Specify zeroes to wait until the completion of data reception.

Example: `LINE INPUT@1,3;A$`

See Also: `CMD DELIM`

ON SRQ GOSUB

Purpose: Statement. Specifies first line of SRQ subroutine.

Syntax: `ON SRQ GOSUB` {*line-no.* | **label*}

Comments: Branches to *line-no.* or **label* when SRQ (service request) has been received in master mode. This statement cannot be used in slave mode.

The `POLL` statement must be used in the subroutine.

This statement can be used only once in all tasks.

Interruption will be stopped right after an `ON SRQ GOSUB` statement is executed. Interruption will stopped in the interruption processing routine.

`RETURN` is used to return to the main routine from the processing routine.

Only a single interruption processing routine can be defined in the tasks. If more than one interruption processing routine is defined, the routine defined last will be effective.

Note
1. If more than one `ON SRQ GOSUB` statement exists in a task, only the last one will be effective.
2. To go to the interruption processing routine, `SRQ` must be ON.
3. `SRQ` is OFF unless an `SRQ ON` statement is executed.

Example: `ON SRQ GOSUB *LABEL`

POLL

Purpose: Statement. Performs serial polling.

Syntax: `POLL` *talker-address*`,` *numeric-variable* [`;` *talker-address*`,` *numeric-variable*]*

Comments: When this statement is executed, UNL (unlisten) command is transmitted followed by SPE (serial poll enable) command.

After that, the talker address in the statement is output, ATN is made false, and the device status transmitted from that talker is received and stored in a *numeric-variable*.

At this time, if the RQS bit (bit 6) of the received device status is ON, it is judged that the device has transmitted the service request. When the device has transmitted the service request, the serial polling ends, and 0 is stored in all the subsequent *numeric-variables*.

The above information is also stored in the IEEE status.

IEEE (4) Device status of the device that has transmitted the service request.

IEEE (5) Talker address of the device that has transmitted the service request.

IEEE (6) Talker address of the device that does not respond to the service request.

Since IEEE (6) will have the Talker address when a IEEE timeout occurs, define a long-enough period by using the `CMD TIMEOUT` statement. For example, `CMD TIMEOUT = 5`.

A GPIB BIOS error will result if `POLL` is executed when the RQS bits (6 bits) of all the devices designated by Talker address are OFF (no SRQ has been issued).

Example: `POLL 3, A; 4, B; 5, C`

PPOLL

Purpose: Statement. Assigns response output line for parallel polling.

Syntax: `PPOLL [PPU] [,` *listener-address*`,` *integer*]*

Comments: This statement is used in the master mode.

PPU: After PPU (Parallel Poll Unconfigure message) has been transmitted, assigns a response output line to the listener. If PPU is omitted, PPU is not transmitted.

Describe the following parameters in hexadecimal number as the *integer*:

Bit	7	6	5	4	3	2	1	0
parallel poll enable (affirmative response)	0	1	1	0	S	P3	P2	P1
parallel poll disable (response stopped)	0	1	1	1	0	0	0	0

S specifies whether a response of 0 or 1 is made to parallel polling.

S	Meanings
0	Responds parallel polling by 0
1	Responds parallel polling by 1

P3, P2, and P1 specify which of DIO1 through DIO8 is used to respond parallel polling.

P3	P2	P1	Meanings
0	0	0	Responds by DIO1
0	0	1	Responds by DIO2
0	1	0	Responds by DIO3
0	1	1	Responds by DIO4
1	0	0	Responds by DIO5
1	0	1	Responds by DIO6
1	1	0	Responds by DIO7
1	1	1	Responds by DIO8

The following table shows the relationship between the set values of the S bit of `PPOLL`, PPR mode (the set value of CMD PPR), and SRQ and the response (DIO1 to DIO8) of the `PPOLL`.

SRQ	OFF		ON	
S bit	0	1	0	1
PPR 0	1	0	1	0

Example: `PPOLL PPU,3,&H6A`

PRINT @

Purpose: Statement. Transmits data as an ASCII character string.

Syntax: `PRINT@` [*listener-address* [, *listener-address*]*]; [*data* [,*data*]*] [@]

Comments: In the master mode, ATN is made true, and the UNL command, MLA (my listen address, and listener address are sequentially transmitted. After that, ATN is made false, and data is transmitted as an ASCII character string.

In the slave mode, the listener address is not specified. The system waits until the talker is addressed by the controller. After MTA (my talk address) has been received, the data is transmitted as an ASCII character string when ATN is made false.

If there is more than one data item, each item is delimited from the others by a comma (,), and the delimiter is transmitted last. If @ is placed at the end of the statement, EOI is made true when the last data byte is output.

Example:

```
PRINT@5,6; "ABC", "DEF" @
```

RBYTE

Purpose: Statement. Receives binary data after transmitting multiline message.

Syntax: RBYTE [*command*] [, *command*]*; [*numeric-variable* [,*numeric-variable*]*

Comments: In the master mode, the command is described as binary data ranging from &H00 to &HFF in hexadecimal notation. After the command has been transmitted with ATN made true, ATN is made false, and the binary data is received and stored in the specified *numeric-variables*.

In the slave mode, the command is not used, and the system waits until addressed as a listener by the controller. When MLA (my listen address) has been received and ATN has been made false, the binary data is received and stored in the *numeric-variables*.

Example:

```
RBYTE &H3F, &H21; ABC%
```

SRQ ON/OFF/STOP

Purpose: Statement. Controls reception of SRQ.

Syntax: SRQ {ON | OFF | STOP}

Comments: This statement is used in the master mode.

SRQ OFF disables interruption of SRQ. When this statement is executed, the execution will not branch to the processing routine defined when SRQ is received.

SRQ ON enables interruption of SRQ. When this statement is executed, the execution will branch to the processing routine when SRQ is received.

SRQ STOP stops interruption of SRQ. When this statement is executed, the execution will not branch to the processing routine when SRQ is received. The reception of SRQ is, however, recorded and the execution will branch to the processing routine when interruption is permitted by SRQ ON.

Note

1. Interruption is stopped right after an ON SRQ GOSUB statement is executed.
2. Interruption will be stopped in the interruption processing routine.

Example:

```
SRQ ON
```

WBYTE

Purpose: Statement. Transmits multiline message and binary data.

Syntax: WBYTE [*command*] [, *command*]*]; [*data* [, *data*]*] [@]

Comments: The *command* and *data* are numbers ranging from &H00 to &HFF in hexadecimal notation.

In the master mode, ATN is made false and the binary data is transmitted after ATN has been made true and the command has been transmitted.

In the slave mode, the command is not used, and the system waits until addressed as a talker by the controller. After MTA (my talk address) has been received, data is transmitted when ATN has been made false.

If @ is placed at the end of the statement, EOI is made true when the last data byte is transmitted.

Example:

```
WBYTE &H3F, &H55; &H31, &H21
```

IEEE

Purpose: Function. Gets status of GP-IB.

Syntax: IEEE(*code*)

Comments: Specify the *code* as a numeric value of 0 to 7 except 3.

IEEE(0) .... Stores the delimiter specified by CMD DELIM as a code number.

Delimiter code	Delimiter
0	CR + LF
1	CR
2	LF
3	EOI

IEEE(1) .... Stores the initial status as the following 8-bit data:

Bit	7	6	5	4	3	2	1	0
data			Mode 0: Master 1: Slave	Own address				

IEEE(2) .... Stores the listener of this Unit, the status of the talker, and the received multiline message as the following bit data:

Bit	Setting and Meaning
8	1: SRQ signal is ON. If this bit is still 1 after the POLL instruction is executed, the next SRQ exists, so POLL must be executed again.
7	1: LLO message is received
6	1: GTL message is received
5	1: DCL or SDC message is received
4	1: GET message is received
3	1: END message (EOI as delimiter) is received
2	1: SPE message is received
1	1: Listener is addressed
0	1: Talker is addressed

IEEE(4) .... Stores the device status of the device that transmits the service request during serial polling.

IEEE(5) .... Stores the talker address of the device that transmits the service request during serial polling.

IEEE(6) .... Stores the talker address of the device that does not respond to the serial polling. 255 will be stored if the device responds. Check this bit when an IEEE Timeout error results in the serial port.

`IEEE(7)` .... Stores the data byte obtained as a result of parallel polling.

Example:

```
PRINT IEEE(0)
```

STATUS

Purpose: Function. Stores device status.

Syntax:

```
STATUS
```

Comments: Stores the device status automatically transmitted in response to serial polling by the controller in the slave mode.

Bit	Meaning
7	General-purpose bit
6	Indicates that SRQ is being transmitted when 1
3 to 0	General-purpose bit

Bit 6 is set by `ISET SRQ` and is reset when serial poll has been received from the controller. The general purpose bit is used to inform the Master of the conditions of the Slaves. Set the data before issuing SRQ.

Example:

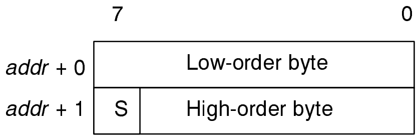
```
STATUS = 2
```

Appendix A
Memory Storage Format of Variables

Variables are stored in the memory as follows depending on their types:

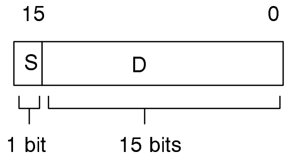
Integer Variable

Integers are stored in 2's-complement format in two consecutive bytes. The low-order byte is at *addr*, and the high-order byte is at *addr+1*.



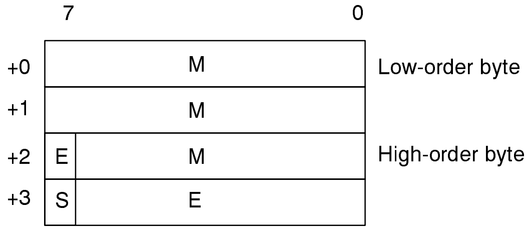
S: sign bit (0: positive, 1: negative)

D: value



Single-precision Floating-point Variable

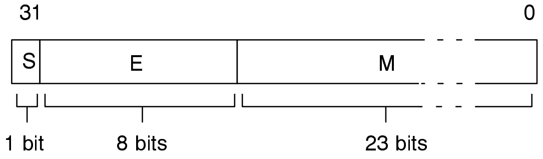
Single-precision floating-point values are stored in IEEE 32-bit format:



S: sign bit (0: positive, 1: negative)

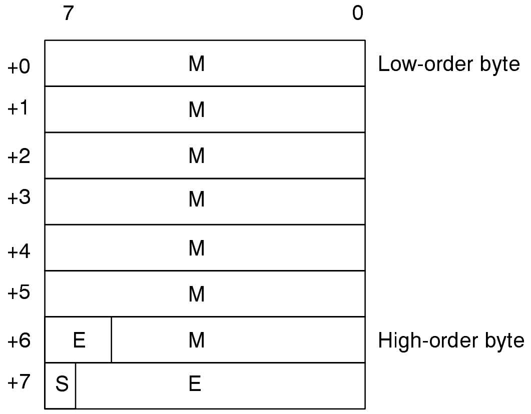
E: exponent (with 127 offsets)

M: mantissa (with MSB always 1)



Double-precision Floating-point Variable

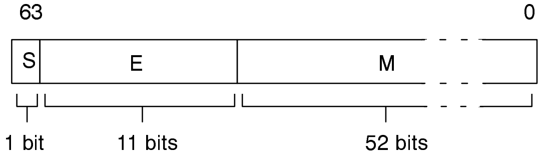
Double-precision floating-point values are stored in IEEE 64-bit format:



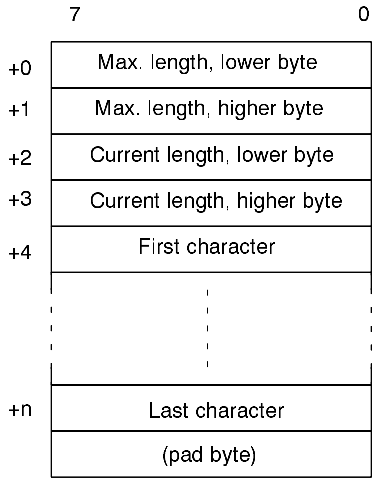
S: sign bit (0: positive, 1: negative)

E: exponent (with 1023 offsets)

M: mantissa (with MSB always 1)



Character Variable

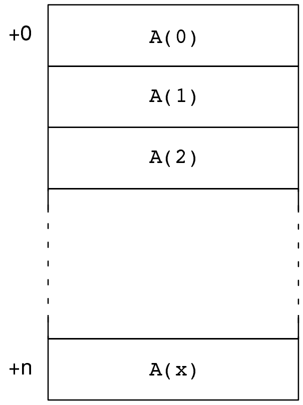


A pad byte may appended if necessary to make the total number of bytes used even.

The value in the pad byte is undefined.

Array Variable

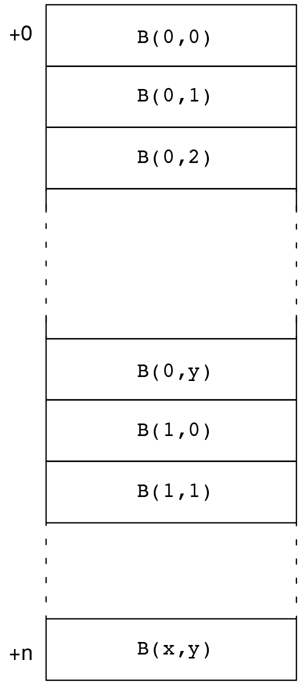
One-dimensional array variable



Each element is stored sequentially in memory. The size of each element is the same as the size of a simple (non-array) variable.

Multi-dimensional Array Variable

Multi-dimensional array variable



The elements of a multi-dimensional array are stored in row-major format; that is, all the elements of one row are stored before the first element of the second row is stored.

Appendix B
BASIC Unit Reserved Words

ABS

ACOS

ALARM ON / OFF / STOP

ASC

ASIN

ATN

AUTO

BITON / BITOFF

BREAK

CALL

CDBL

CHR$

CINT

CLOSE

CLS

CMD DELIM

CMD PPR

CMD TIMEOUT

COM ON / OFF / STOP

CONT

COS

CSNG

CVI / CVS / CVD

DATA

DATE$

DEF FN

DEF USR

DEFINT / DEFSNG/ DEFDBL / DEFSTR

DEG SEG

DELETE

DIM

EDIT

END

END PARACT

EOF

ERL/ERR

ERROR

EXIT

EXP

FIELD

FILES / LFILES

FINS ON / OFF / STOP

FIX

FOR... TO... STEP...

FRE

GET

GOSUB / RETURN

GOTO

HEX$

IEEE(0)

IEEE(1)

IEEE(2)

IEEE(4)

IEEE(5)

IEEE(6)

IEEE(7)

IF... GOTO... ELSE...

INKEY$

INPUT

INPUT #

INPUT @

INPUT$

INSTR

INT

INTRB

INTRL

INTRR

IRESET REN

ISET IFC

ISET REN

ISET SRQ

KEY ON / OFF / STOP

KILL

LEFT$

LEN

LET

LINE INPUT

LINE INPUT #

LINE INPUT @

LIST / LLIST

LOAD

LOC

LOCATE

LOF

LOG

LPRINT

LPRINT USING

LSET/RSET

MERGE

MESSAGE

MID$

MKI$ / MKS$ / MKD$

MON

MSET

NAME

NEW

NEXT

OCT$

ON ALARM GOSUB

ON COM GOSUB

ON ERROR GOTO

ON FINS GOSUB

ON GOSUB

ON GOTO

ON KEY GOSUB

ON PC GOSUB

ON SIGNAL GOSUB

ON SRQ GOSUB

ON TIME$ GOSUB

ON TIMER GOSUB

OPEN

OPTION BASE

OPTION ERASE

OPTION LENGTH

PARACT

PAUSE

PC ON / OFF / STOP

PC READ

PC WRITE

PEEK

PGEN

PINF

PNAME

POKE

POLL

PPOLL

PRINT #

PRINT # USING

PRINT / ?

PRINT @

PRINT USING

PUT

RANDOMIZE

RBYTE

RDIM

READ

RECEIVE

REM

RENUM

RESTORE

RESUME

RIGHT$

RND

ROMLOAD

ROMSAVE

ROMVERIFY

RUN

RUNr

SAVE

SEARCH

SEND

SENDSIG

SGN

SIGNAL ON / OFF / STOP

SIN

SPACE$

SPC

SQR

SRQ ON/OFF/STOP

STATUS

STEP

STOP

STR$

STRING$

SWAP

TAB

TAN

TASK

TIME$

TIME$ ON / OFF / STOP

TIMER ON / OFF / STOP

TROFF

TRON

TWAIT

USR

VAL

VARPTR

VERIFY

VLOAD

VSAVE

WBYTE

WHILE/WEND

WRITE

WRITE #

Appendix C
Extended ASCII

Programming Console and Data Access Console

The first two columns (HEX 0 and 1) are displayed when in the control code display mode.

Bits 0 to 3 (lower bits)		Bits 4 to 7 (upper bits)													
BIN		**0000**	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**	**1010**	**1011**	**1100**	**1101**	**1110**	**1111**
	HEX	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**A**	**B**	**C**	**D**	**E**	**F**
0000	**0**	NUL	DLE	Space	0	@	P	‘	p		0	@	P	‘	p
0001	**1**	SOH	DC$_1$	!	1	A	Q	a	q	!	1	A	Q	a	q
0010	**2**	STX	DC$_2$	”	2	B	R	b	r	”	2	B	R	b	r
0011	**3**	ETX	DC$_3$	#	3	C	S	c	s	#	3	C	S	c	s
0100	**4**	EOT	DC$_4$	$	4	D	T	d	t	$	4	D	T	d	t
0101	**5**	ENQ	NAK	%	5	E	U	e	u	%	5	E	U	e	u
0110	**6**	ACK	SYN	&	6	F	V	f	v	&	6	F	V	f	v
0111	**7**	BEL	ETB	’	7	G	W	g	w	’	7	G	W	g	w
1000	**8**	BS	CAN	(	8	H	X	h	x	(	8	H	X	h	x
1001	**9**	HT	EM	)	9	I	Y	i	y	)	9	I	Y	i	y
1010	**A**	LF	SUB	*	:	J	Z	j	z	*	:	J	Z	j	z
1011	**B**	VT	ESC	+	;	K	[	k	{	+	;	K	[	k	{
1100	**C**	FF	FS	,	<	L	\	l	|	,	<	L	\	l	|
1101	**D**	CR	GS	-	=	M	]	m	}	-	=	M	]	m	}
1110	**E**	S0	RS	.	>	N	^	n	«	.	>	N	^	n	
1111	**F**	S1	US	/	?	O	_	o	~	/	?	O	_	o	~

Glossary

active controller The device on a general-purpose interface bus that is currently controlling communications on the bus.

address A number used to identify the location of data or programming instructions in memory or to identify the location of a network or a unit in a network.

address command A command sent to a specific address on a general-purpose interface bus.

advanced instruction An instruction input with a function code that handles data processing operations within ladder diagrams, as opposed to a basic instruction, which makes up the fundamental portion of a ladder diagram.

allocation The process by which the PC assigns certain bits or words in memory for various functions. This includes pairing I/O bits to I/O points on Units.

alphanumeric character An upper- or lower-case letter, digit, or underscore (_). The underscore is considered to be a letter.

analog Something that represents or can process a continuous range of values as opposed to values that can be represented in distinct increments. Something that represents or can process values represented in distinct increments is called digital.

Analog I/O Unit I/O Units that convert I/O between analog and digital values. An Analog Input Input converts an analog input to a digital value for processing by the PC. An Analog Output Unit converts a digital value to an analog output.

AND A logic operation whereby the result is true if and only if both premises are true. In ladder-diagram programming the premises are usually ON/OFF states of bits or the logical combination of such states called execution conditions.

area See *data area* and *memory area*.

area prefix A one or two letter prefix used to identify a memory area in the PC. All memory areas except the CIO area require prefixes to identify addresses in them.

argument A value passed to a function when the function is called.

arithmetic operator A character indicating to the BASIC Unit that it should perform some sort of calculation; for instance, "+" indicates addition, and "*" indicates multiplication.

array element One part of an array variable. An array element can be another array (for multi-dimensional arrays) or a simple variable (an integer, floating-point, string, etc.)

array subscript An integer expression used to designate an array element for some operation.

array variable A variable which consists of a collection of parts called array elements. Each element can be another array (for multi-dimensional arrays) or a simple variable (an integer, floating-point, string, etc.)

ASCII Short for American Standard Code for Information Interchange. ASCII is used to code characters for output to printers and other external devices.

assembler A program which converts machine-language mnemonics to machine instructions.

asynchronous execution Execution of programs and servicing operations in which program execution and servicing are not synchronized with each other.

Auxiliary Area A PC data area allocated to flags and control bits.

auxiliary bit A bit in the Auxiliary Area.

back-up A copy made of existing data to ensure that the data will not be lost even if the original data is corrupted or erased.

BASIC A common programming language. BASIC Units are programmed in BASIC.

basic instruction A fundamental instruction used in a ladder diagram. See *advanced instruction*.

BASIC Unit A CPU Bus Unit used to run programs in BASIC.

baud rate The data transmission speed between two devices in a system measured in bits per second.

BCD Short for binary-coded decimal.

binary A number system where all numbers are expressed in base 2, i.e., numbers are written using only 0's and 1's. Each group of four binary bits is equivalent to one hexadecimal digit. Binary data in memory is thus often expressed in hexadecimal for convenience.

binary-coded decimal A system used to represent numbers so that every four binary bits is numerically equivalent to one decimal digit.

bit The smallest piece of information that can be represented on a computer. A bit has the value of either zero or one, corresponding to the electrical signals ON and OFF. A bit represents one binary digit. Some bits at particular addresses are allocated to special purposes, such as holding the status of input from external devices, while other bits are available for general use in programming.

bit address The location in memory where a bit of data is stored. A bit address specifies the data area and word that is being addressed as well as the number of the bit within the word.

breakpoint Used during program debugging to mark places where the BASIC Unit should stop executing the program and allow the programmer to check the state of the program's variables.

buffer A temporary storage space for data in a computerized device.

building-block PC A PC that is constructed from individual components, or “building blocks.” With building-block PCs, there is no one Unit that is independently identifiable as a PC. The PC is rather a functional assembly of Units.

bus A communications path used to pass data between any of the Units connected to it.

bus link A data link that passed data between two Units across a bus.

byte A unit of data equivalent to 8 bits, i.e., half a word.

central processing unit A device that is capable of storing programs and data, and executing the instructions contained in the programs. In a PC System, the central processing unit executes the program, processes I/O signals, communicates with external devices, etc.

channel See *word*.

character code A numeric (usually binary) code used to represent an alphanumeric character.

character constant A character expression which contains no string variables.

character expression An expression involving only character strings, string variables, functions returning character strings, and the "+" operator.

character string A sequence of characters delimited by double quotes (`"`).

checksum A sum transmitted with a data pack in communications. The checksum can be recalculated from the received data to confirm that the data in the transmission has not been corrupted.

CIO Area A memory area used to control I/O and to store and manipulate data. CIO Area addresses do not require prefixes.

command A BASIC Unit instruction which is usually used in immediate mode (e.g. `LIST`, `RUN`, or `NEW`).

command format The syntax required for use in a command and specifying what data is required in what order.

comment statement A statement which is ignored by the BASIC Unit. They may be included in a program to describe the program or to explain how it is supposed to work. Lines beginning with the `REM` instruction are comments, and the single quote character (`'`) begins a comment which extends to the end of the current line.

communications port interrupt An interrupt that occurs when a character is received by one of the communications ports.

constant An input for an operand in which the actual numeric value is specified. Constants can be input for certain operands in place of memory area addresses. Some operands must be input as constants.

control bit A bit in a memory area that is set either through the program or via a Programming Device to achieve a specific purpose, e.g., a Restart Bit is turned ON and OFF to restart a Unit.

control signal A signal sent from the PC to effect the operation of the controlled system.

Control System All of the hardware and software components used to control other devices. A Control System includes the PC System, the PC programs, and all I/O devices that are used to control or obtain feedback from the controlled system.

controlled system The devices that are being controlled by a PC System.

controller A device on a general-purpose interface bus that is capable of controlling communications.

CPU See *central processing unit*.

CPU Bus Unit
A special Unit used with CV-series PCs that mounts to the CPU bus. This connection to the CPU bus enables special data links, data transfers, and processing.

CPU Rack
The main Rack in a building-block PC, the CPU Rack contains the CPU, a Power Supply, and other Units. The CPU Rack, along with the Expansion CPU Rack, provides both an I/O bus and a CPU bus.

C-series PC
Any of the following PCs: C2000H, C1000H, C500, C200H, C40H, C28H, C20H, C60K, C60P, C40K, C40P, C28K, C28P, C20K, C20P, C120, or C20.

CTS signal
A signal used in communications between electronic devices to indicate that the receiver is ready to accept incoming data.

CV Support Software
A programming package run on an IBM PC/AT or compatible to serve as a Programming Device for CV-series PCs.

CV-series PC
Any of the following PCs: CV500, CV1000, CV2000, or CVM1.

CVSS
See *CV Support Software*.

cycle
One unit of processing performed by the CPU, including SFC/ladder program execution, peripheral servicing, I/O refreshing, etc. The cycle is called the scan with C-series PCs.

cycle time
The time required to complete one cycle of CPU processing.

cyclic (data) transfer
A transfer of data that occurs at a specific interval.

DAC
See *Data Access Console*.

Data Access Console
A Programming Device used to monitor and control memory area contents. The Data Access Console does not afford the wide range of programming capabilities as the GPC or CVSS and is designed for system monitoring and maintenance.

data area
An area in the PC's memory that is designed to hold a specific type of data.

data link
An automatic data transmission operation that allows PCs or Units within PC to pass data back and forth via common data areas.

data register
A storage location in memory used to hold data. In CV-series PCs, data registers are used with or without index registers to hold data used in indirect addressing.

data transfer
Moving data from one memory location to another, either within the same device or between different devices connected via a communications line or network.

debug
A process by which a draft program is corrected until it operates as intended. Debugging includes both the removal of syntax errors, as well as the fine-tuning of timing and coordination of control operations.

decimal
A number system where numbers are expressed to the base 10. In a PC all data is ultimately stored in binary form, four binary bits are often used to represent one decimal digit, via a system called binary-coded decimal.

decimal integer constant
An integer constant expressed in decimal notation. Such a constant uses only the numerals 0 through 9.

declarator A special character added to a variable to specify the type of variable, e.g., a character, a single-precision real number, etc.

decrement Decreasing a numeric value, usually by 1.

default A value automatically set by the PC when the user does not specifically set another value. Many devices will assume such default conditions upon the application of power.

destination The location where an instruction places the data on which it is operating, as opposed to the location from which data is taken for use in the instruction. The location from which data is taken is called the source.

destination line The target of a `GOTO` or `GOSUB` statement.

destination variable The variable which is to receive the results of a calculation or operation (the variable in which the results are to be stored).

digit A unit of storage in memory that consists of four bits.

DIP switch Dual in-line package switch, an array of pins in a signal package that is mounted to a circuit board and is used to set operating parameters.

distributed control A automation concept in which control of each portion of an automated system is located near the devices actually being controlled, i.e., control is decentralized and ‘distributed’ over the system. Distributed control is a concept basic to PC Systems.

DM Area A data area used to hold only word data. Words in the DM area cannot be accessed bit by bit.

DM word A word in the DM Area.

double-precision constant A floating-point constant which has at least one of these properties: a trailing hash mark (e.g. `123.45#`); an exponent declared with `D` or `d` instead of `E` or `e` (e.g. `1.2345D2`); or more than 15 digits in the mantissa (e.g. `123.450000000000`).

double-precision variable A variable which can hold a double-precision value.

downloading The process of transferring a program or data from a higher-level or host computer to a lower-level or slave computer. If a Programming Device is involved, the Programming Device is considered the host computer.

DSR signal Data Set Ready signal; sent by a modem to indicate that it is functional.

EEPROM Electrically erasable programmable read-only memory; a type of ROM in which stored data can be erased and reprogrammed. This is accomplished using a special control lead connected to the EEPROM chip and can be done without having to remove the EEPROM chip from the device in which it is mounted.

elapsed-time interrupt An interrupt which occurs after a specified period of time.

electrical noise Random variations of one or more electrical characteristics such as voltage, current, and data, which might interfere with the normal operation of a device.

EM Area Extended Data Memory Area; an area that can be optionally added to certain PCs to enable greater data storage. Functionally, the EM Area operates like the

DM Area. Area addresses are prefixes with E and only words can be accessed. The EM Area is separated into multiple banks.

EPROM Erasable programmable read-only memory; a type of ROM in which stored data can be erased, by ultraviolet light or other means, and reprogrammed.

error code A numeric code generated to indicate that an error exists, and something about the nature of the error. Some error codes are generated by the system; others are defined in the program by the operator.

error generation number A number used to identify an error generated by a program.

event (data) transfer A data transfer that is performed in response to an event, e.g., an interrupt signal.

event processing Processing that is performed in response to an event, e.g., an interrupt signal.

executable statement A statement which causes the BASIC Unit to perform some operation, rather than one which changes the way the BASIC Unit interprets the program. (For example, `PRINT` is an executable statement, but `REM` is not.)

Expansion CPU Rack A Rack connected to the CPU Rack to increase the virtual size of the CPU Rack. Units that may be mounted to the CPU Backplane may also be mounted to the Expansion CPU Backplane.

Expansion I/O Rack A Rack used to increase the I/O capacity of a PC. In CV-Series PC, either one Expansion I/O Rack can be connected directly to the CPU or Expansion CPU Rack or multiple Expansion I/O Racks can be connected by using an I/O Control and I/O Interface Units.

expression The translation of a mathematical formula into BASIC notation. For example, the formula for the area of a circle is: $A=\pi r^2$; the BASIC expression to calculate the area of a circle is: `AREA=3.1415*RADIUS^2`.

FA Factory automation.

factory computer A general-purpose computer, usually quite similar to a business computer, that is used in automated factory control.

fatal error An error that stops PC operation and requires correction before operation can continue.

FINS See *CV-mode*.

flag A dedicated bit in memory that is set by the system to indicate some type of operating status. Some flags, such as the carry flag, can also be set by the operator or via the program.

floating-point decimal A decimal number expressed as a number (the mantissa) multiplied by a power of 10, e.g., 0.538 x 10^{-5}.

floating-point format The layout of a single- or double-precision value in memory.

floating-point constant A numeric constant which has a fractional or exponential part.

force reset The process of forcibly turning OFF a bit via a programming device. Bits are usually turned OFF as a result of program execution.

force set The process of forcibly turning ON a bit via a programming device. Bits are usually turned ON as a result of program execution.

frame checksum The results of exclusive ORing all data within a specified calculation range. The frame checksum can be calculated on both the sending and receiving end of a data transfer to confirm that data was transmitted correctly.

function A BASIC Unit instruction which calculates a value based on its arguments and returns the value to the program. The programmer can define new functions with the `DEF FN` statement.

general-purpose interface bus A bus used to connect various devices to a computer.

generation line The line in a program that generates an event, e.g., an interrupt.

global variable A variable which can be accessed from any of the tasks in a program.

GPC An acronym for Graphic Programming Console.

GP-IB An acronym for general-purpose interface bus.

Graphic Programming Console A programming device with advanced programming and debugging capabilities to facilitate PC operation. A Graphic Programming Console is provided with a large display onto which ladder-diagram programs can be written directly in ladder-diagram symbols for input into the PC without conversion to mnemonic form.

handshake line A line in a program or a physical connection between devices used for handshaking.

handshaking The process whereby two devices exchange basic signals to coordinate communications between them.

hexadecimal A number system where all numbers are expressed to the base 16. In a PC all data is ultimately stored in binary form, however, displays and inputs on Programming Devices are often expressed in hexadecimal to simplify operation. Each group of four binary bits is numerically equivalent to one hexadecimal digit.

hexadecimal constant An integer constant expressed in hexadecimal notation. Hexadecimal constants must begin with the characters `&H` or `&h` and contain only hexadecimal digits (numerals `0` through `9` and letters `a` through `f` or `A` through `F`).

host interface An interface that allows communications with a host computer.

Host Link System A system with one or more host computers connected to one or more PCs via Host Link Units or host interfaces so that the host computer can be used to transfer data to and from the PC(s). Host Link Systems enable centralized management and control of PC Systems.

Host Link Unit An interface used to connect a C-series PC to a host computer in a Host Link System.

I/O allocation The process by which the PC assigns certain bits in memory for various functions. This includes pairing I/O bits to I/O points on Units.

I/O Block Either an Input Block or an Output Block. I/O Blocks provide mounting positions for replaceable relays.

I/O Control Unit
A Unit mounted to the CPU Rack to monitor and control I/O points on Expansion CPU Racks or Expansion I/O Racks.

I/O delay
The delay in time from when a signal is sent to an output to when the status of the output is actually in effect or the delay in time from when the status of an input changes until the signal indicating the change in the status is received.

I/O device
A device connected to the I/O terminals on I/O Units, Special I/O Units, etc. I/O devices may be either part of the Control System, if they function to help control other devices, or they may be part of the controlled system.

I/O Interface Unit
A Unit mounted to an Expansion CPU Rack or Expansion I/O Rack to interface the Rack to the CPU Rack.

I/O point
The place at which an input signal enters the PC System, or at which an output signal leaves the PC System. In physical terms, I/O points correspond to terminals or connector pins on a Unit; in terms of programming, an I/O points correspond to I/O bits in the IR area.

I/O refreshing
The process of updating output status sent to external devices so that it agrees with the status of output bits held in memory and of updating input bits in memory so that they agree with the status of inputs from external devices.

I/O response time
The time required for an output signal to be sent from the PC in response to an input signal received from an external device.

I/O Terminal
A Remote I/O Unit connected in a Wired Remote I/O System to provide a limited number of I/O points at one location. There are several types of I/O Terminals.

I/O Unit
The most basic type of Unit mounted to a Backplane. I/O Units include Input Units and Output Units, each of which is available in a range of specifications. I/O Units do not include Special I/O Units, Link Units, etc.

I/O verification error
A error generated by a disagreement between the Units registered in the I/O table and the Units actually mounted to the PC.

I/O word
A word in the CIO area that is allocated to a Unit in the PC System and is used to hold I/O status for that Unit.

IBM PC/AT or compatible
A computer that has similar architecture to, that is logically compatible with, and that can run software designed for an IBM PC/AT computer.

initialize
Part of the startup process whereby some memory areas are cleared, system setup is checked, and default values are set.

input
The signal coming from an external device into the PC. The term input is often used abstractly or collectively to refer to incoming signals.

input bit
A bit in the CIO area that is allocated to hold the status of an input.

Input Block
A Unit used in combination with a Remote Interface to create an I/O Terminal. An Input Block provides mounting positions for replaceable relays. Each relay can be selected according to specific input requirements.

input device
An external device that sends signals into the PC System.

input point
The point at which an input enters the PC System. Input points correspond physically to terminals or connector pins.

input signal A change in the status of a connection entering the PC. Generally an input signal is said to exist when, for example, a connection point goes from low to high voltage or from a nonconductive to a conductive state.

Input Terminal An I/O Terminal that provides input points.

instruction A direction given in the program that tells the PC of the action to be carried out, and the data to be used in carrying out the action. Instructions can be used to simply turn a bit ON or OFF, or they can perform much more complex actions, such as converting and/or transferring large blocks of data.

integer constant A numeric value which has a percent sign (`%`) appended, or an expression containing only integer constants.

integer variable A variable that can hold an integer value.

Intel HEX record Hexadecimal data recorded according to the Intel standard.

Intelligent Signal Processor A control-panel interface used to access and control signals. The Processor is capable of processing the signals according to specifications, and thus the name.

interface An interface is the conceptual boundary between systems or devices and usually involves changes in the way the communicated data is represented. Interface devices such as NSBs perform operations like changing the coding, format, or speed of the data.

interrupt (signal) A signal that stops normal program execution and causes a subroutine to be run or other processing to take place.

Interrupt Input Unit A Rack-mounting Unit used to input external interrupts into a PC System.

interrupt service routine A BASIC subroutine which is called in response to an interrupt.

inter-task communication Communication (transfer of data or status information) between two tasks in a BASIC Unit program.

interval interrupt An interrupt which occurs each time a certain amount of time has elapsed.

IOIF An acronym for I/O Interface Unit.

IOM (Area) A collective memory area containing all of the memory areas that can be accessed by bit, including timer and counter Completion Flags. The IOM Area includes all memory area memory addresses between 0000 and 0FFF.

JIS An acronym for Japanese Industrial Standards.

jump A type of programming where execution moves directly from one point in a program to another, without sequentially executing any instructions in between. Jumps in ladder diagrams are usually conditional on an execution condition; jumps in SFC programs are conditional on the step status and transition condition status before the jump.

keyword A word that has special meaning to the BASIC Unit. Programs cannot use keywords for variable or label names.

label A name attached to a program line for use in `GOTO` and `GOSUB` statements.

least-significant (bit/word) See *rightmost (bit/word)*.

LED Acronym for light-emitting diode; a device used as for indicators or displays.

leftmost (bit/word) The highest numbered bits of a group of bits, generally of an entire word, or the highest numbered words of a group of words. These bits/words are often called most-significant bits/words.

line number An integer which uniquely identifies a line within a program. Line numbers may be used in `GOTO` and `GOSUB` statements.

line One portion of a BASIC program. A line consists of a line number and one or more statements.

link A hardware or software connection formed between two Units. “Link” can refer either to a part of the physical connection between two Units or a software connection created to data existing at another location (i.e., data links).

Link System A system used to connect remote I/O or to connect multiple PCs in a network. Link Systems include the following: SYSMAC BUS Remote I/O Systems, SYSMAC BUS/2 Remote I/O Systems, SYSMAC LINK Systems, Host Link Systems, and SYSMAC NET Link Systems.

Link Unit Any of the Units used to connect a PC to a Link System. These include Remote I/O Units, SYSMAC LINK Units, and SYSMAC NET Link Units.

listener A device on a general-purpose interface bus that is receiving data from another device on the bus.

listener address The addresses on a general-purpose interface bus of a device that is receiving data from another device on the bus.

load The processes of copying data either from an external device or from a storage area to an active portion of the system such as a display buffer. Also, an output device connected to the PC is called a load.

local variable A variable which can only be accessed by the task in which it is declared.

logical expression An expression made up of one or more logical operations, which has “TRUE” or “FALSE” as its value.

logical operation An operation on one or more “TRUE” or “FALSE” values (a Boolean operation), or an operation which returns a “TRUE” or “FALSE” indication.

logical operator A keyword or symbol which instructs the BASIC Unit to perform some calculation that returns a “TRUE” or “FALSE” value.

loop A group of instructions that can be executed more than once in succession (i.e., repeated) depending on an execution condition or bit status.

LSI An acronym for large scale integration.

machine code The binary program code that is actual executed by a CPU.

machine language A programming language in which the program is written directly into machine code.

MCR Unit Magnetic Card Reader Unit.

megabyte A unit of storage equal to one million bytes.

memory area Any of the areas in the PC used to hold data or programs.

memory switch A bit or bits in memory that are used to set operating parameters similar to the way a hardware switch would be.

most-significant (bit/word) See *leftmost (bit/word).*

Motorola S-record A format standardized by the Motorola company to store programs.

MS-DOS An operating system in common use on smaller computers.

multi-dimensional array An array in which more than one subscript is required to access an element.

multidrop configuration A bus configuration in which all devices are connected in series, but across, not through, each device.

multitasked program A program which consists of two or more sub-programs or “tasks” executing concurrently.

multitasking Describes a computer which can run more than one program at a time, or which can give the illusion that several programs are running simultaneously.

my-address The address of a device on a general-purpose interface bus.

nesting Programming one loop within another loop, programming a call to a subroutine within another subroutine, or programming an IF-ELSE programming section within another IF-ELSE section.

network interrupt An interrupt that occurs when data is received on the network interface.

Network Service Board A device with an interface to connect devices other than PCs to a SYSMAC NET Link System.

Network Service Unit A Unit that provides two interfaces to connect peripheral devices to a SYSMAC NET Link System.

noise interference Disturbances in signals caused by electrical noise.

non-executable statement A statement that changes the way the BASIC Unit processes the program, but does not cause the Unit to perform any particular operation. For example, the `REM` statement causes the Unit to ignore the rest of the line.

nonfatal error A hardware or software error that produces a warning but does not stop the PC from operating.

non-volatile variable A variable that is stored in battery-backed memory. Non-volatile variables retain their values even if power to the Unit is turned off.

NOT A logic operation which inverts the status of the operand. For example, AND NOT indicates an AND operation with the opposite of the actual status of the operand bit.

null string A string containing no characters (`""`).

numeric constant A number (integer or floating-point) or a numeric expression containing no variables or function calls.

numeric expression A sequence of numbers, variables, and arithmetic operators that instructs the BASIC Unit to calculate a numeric value.

numeric key interrupt An interrupt that occurs when the user presses one of the numeric keypad keys.

numeric variable A variable that can hold a numeric value.

object code The code that a program is converted to before actual execution. See *source code*.

octal A number system where all numbers are expressed in base 8, i.e., numbers are written using only numerals 0 through 7.

octal constant An integer constant expressed in octal notation. Octal constants must begin with `&`, `&O`, or `&o` and contain only octal digits (numerals `0` through `7`).

OFF The status of an input or output when a signal is said not to be present. The OFF state is generally represented by a low voltage or by non-conductivity, but can be defined as the opposite of either.

OFF delay The delay between the time when a signal is switched OFF (e.g., by an input device or PC) and the time when the signal reaches a state readable as an OFF signal (i.e., as no signal) by a receiving party (e.g., output device or PC).

offset A positive or negative value added to a base value such as an address to specify a desired value.

ON The status of an input or output when a signal is said to be present. The ON state is generally represented by a high voltage or by conductivity, but can be defined as the opposite of either.

ON delay The delay between the time when an ON signal is initiated (e.g., by an input device or PC) and the time when the signal reaches a state readable as an ON signal by a receiving party (e.g., output device or PC).

operand The values designated as the data to be used for an instruction. An operand can be input as a constant expressing the actual numeric value to be used or as an address to express the location in memory of the data to be used.

operating error An error that occurs during actual PC operation as opposed to an initialization error, which occurs before actual operations can begin.

operator A character that instructs the BASIC Unit to perform some calculation. For example, the "+" character indicates that the BASIC Unit should add two numeric values (or concatenate two strings).

operator priority Controls the order of evaluation for sub-expressions in a numeric expression. For example, `2+3*4` is interpreted as `2+(3*4)` or 14 (and not `(2+3)*4` or 20), because the operator priority for `*` is higher than that for `+`. Parentheses may be used to change the order in which sub-expressions are evaluated.

OR A logic operation whereby the result is true if either of two premises is true, or if both are true. In ladder-diagram programming the premises are usually ON/OFF states of bits or the logical combination of such states called execution conditions.

OS Operating system; the basic software the drives a computer and on which all other software is executed.

output The signal sent from the PC to an external device. The term output is often used abstractly or collectively to refer to outgoing signals.

Output Block A Unit used in combination with a Remote Interface to create an I/O Terminal. An Output Block provides mounting positions for replaceable relays. Each relay can be selected according to specific output requirements.

output device An external device that receives signals from the PC System.

output point The point at which an output leaves the PC System. Output points correspond physically to terminals or connector pins.

output signal A signal being sent to an external device. Generally an output signal is said to exist when, for example, a connection point goes from low to high voltage or from a nonconductive to a conductive state.

Output Terminal An I/O Terminal that provides output points.

overflow The state where the capacity of a data storage location has been exceeded.

overwrite Changing the content of a memory location so that the previous content is lost.

pad byte An extra byte added at the end of a string to make the total number of characters in the string even.

parallel polling A polling method in which all devices in a system are polled at the same time.

parity Adjustment of the number of ON bits in a word or other unit of data so that the total is always an even number or always an odd number. Parity is generally used to check the accuracy of data after being transmitted by confirming that the number of ON bits is still even or still odd.

parity check Checking parity to ensure that transmitted data has not been corrupted.

PC An acronym for Programmable Controller.

PC configuration The arrangement and interconnections of the Units that are put together to form a functional PC.

PC System With building-block PCs, all of the Racks and independent Units connected directly to them up to, but not including the I/O devices. The boundaries of a PC System are the PC and the program in its CPU at the upper end; and the I/O Units, Special I/O Units, Optical I/O Units, Remote Terminals, etc., at the lower end.

PCB An acronym for printed circuit board.

PC Setup A group of operating parameters set in the PC from a Programming Device to control PC operation.

Peripheral Device Devices connected to a PC System to aid in system operation. Peripheral devices include printers, programming devices, external storage media, etc.

peripheral servicing Processing signals to and from peripheral devices, including refreshing, communications processing, interrupts, etc.

PID Unit A Unit designed for PID control.

placeholder A zero that is required to indicate the place value of other digits in a numeral, e.g., the zeros to the right of the decimal point in the following number: 0.0045.

pointer
A variable or register which contains the address of some object in memory.

present value
The current value registered in a device at any instant during its operation. Present value is abbreviated as PV. The use of this term is generally restricted to timers and counters.

printed circuit board
A board onto which electrical circuits are printed for mounting into a computer or electrical device.

program code
The representation of a program used internally by the BASIC Unit.

Programmable Controller
A computerized device that can accept inputs from external devices and generate outputs to external devices according to a program held in memory. Programmable Controllers are used to automate control of external devices. Although single-unit Programmable Controllers are available, building-block Programmable Controllers are constructed from separate components. Such Programmable Controllers are formed only when enough of these separate components are assembled to form a functional assembly, i.e., there is no one individual Unit called a PC.

Programming Console
The simplest form or programming device available for a PC. Programming Consoles are available both as hand-held models and as CPU-mounting models.

Programming Device
A Peripheral Device used to input a program into a PC or to alter or monitor a program already held in the PC. There are dedicated programming devices, such as Programming Consoles, and there are non-dedicated devices, such as a host computer.

PROM
Programmable read-only memory; a type of ROM into which the program or data may be written after manufacture, by a customer, but which is fixed from that time on.

PROM Writer
A peripheral device used to write programs and other data into a ROM for permanent storage and application.

prompt
A message or symbol that appears on a display to request input from the operator.

protocol
The parameters and procedures that are standardized to enable two devices to communicate or to enable a programmer or operator to communicate with a device.

PV
See *present value*.

Rack
An assembly that forms a functional unit in a Rack PC System. A Rack consists of a Backplane and the Units mounted to it. These Units include the Power Supply, CPU, and I/O Units. Racks include CPU Racks, Expansion I/O Racks, and I/O Racks. The CPU Rack is the Rack with the CPU mounted to it. An Expansion I/O Rack is an additional Rack that holds extra I/O Units. An I/O Rack is used in the C2000H Duplex System, because there is no room for any I/O Units on the CPU Rack in this System.

rack number
A number assigned to a Rack according to the order that it is connected to the CPU Rack, with the CPU Rack generally being rack number 0.

Rack PC
A PC that is composed of Units mounted to one or more Racks. This configuration is the most flexible, and most large PCs are Rack PCs. A Rack PC is the

opposite of a Package-type PC, which has all of the basic I/O, storage, and control functions built into a single package.

RAM Random access memory; a data storage media. RAM will not retain data when power is disconnected.

random access file A file that can be accessed at any desired point, and not only sequentially.

RAS An acronym for reliability, assurance, safety.

record One block or unit of data in a sequential access file.

refresh The process of updating output status sent to external devices so that it agrees with the status of output bits held in memory and of updating input bits in memory so that they agree with the status of inputs from external devices.

register A special memory location inside the BASIC Unit's CPU.

relative expression A logical expression concerning the magnitudes of two numeric or string expressions (for example, `A>B` is a relative expression which is TRUE if the value of `A` is greater than the value of `B`, and FALSE otherwise).

relative operator A character (e.g. >, <, `=`) or pair of characters (e.g. >=, <=) used in a relative expression.

relay-based control The forerunner of PCs. In relay-based control, groups of relays are interconnected to form control circuits. In a PC, these are replaced by programmable circuits.

reserved bit A bit that is not available for user application.

reserved word A word in memory that is reserved for a special purpose and cannot be accessed by the user.

reset The process of turning a bit or signal OFF or of changing the present value of a timer or counter to its set value or to zero.

Restart Bit A bit used to restart a Unit mounted to a PC.

restart continuation A process which allows memory and program execution status to be maintained so that PC operation can be restarted from the state it was in when operation was stopped by a power interruption.

retrieve The processes of copying data either from an external device or from a storage area to an active portion of the system such as a display buffer. Also, an output device connected to the PC is called a load.

retry The process whereby a device will re-transmit data which has resulted in an error message from the receiving device.

rightmost (bit/word) The lowest numbered bits of a group of bits, generally of an entire word, or the lowest numbered words of a group of words. These bits/words are often called least-significant bits/words.

rising edge The point where a signal actually changes from an OFF to an ON status.

ROM Read only memory; a type of digital storage that cannot be written to. A ROM chip is manufactured with its program or data already stored in it and can never be changed. However, the program or data can be read as many times as desired.

round-robin In order, completing one item before moving on to the next.

routine A section of a program; often one which may be called by other parts of the program as a subroutine.

row-major form Describes the layout of the elements of an array variable in memory.

RS-232C interface An industry standard for serial communications.

RS-422 interface An industry standard for serial communications.

RTS signal Request To Send: the BASIC Unit can be programmed to assert this signal when it wishes to send data through a communications port.

scan The process used to execute a ladder-diagram program. The program is examined sequentially from start to finish and each instruction is executed in turn based on execution conditions. The scan also includes peripheral processing, I/O refreshing, etc. The scan is called the cycle with CV-series PCs.

scan time The time required for a single scan of a ladder-diagram program.

secondary command A command sent with a listener address to specify the address of another listener or talker.

segment A 64K-byte block of memory beginning on a 16-byte boundary. The BASIC Unit's CPU has several registers that can hold the address of the beginning of a segment.

self diagnosis A process whereby the system checks its own operation and generates a warning or error if an abnormality is discovered.

sequential access file A file that can be read or written only sequential from the beginning to the end.

serial polling A polling method in which each device being polled is polled one at a time in sequence.

series A wiring method in which Units are wired consecutively in a string. In Link Systems wired through Link Adapters, the Units are still functionally wired in series, even though Units are placed on branch lines.

service request A signal from a device requesting that some sort of processing occur.

servicing The process whereby the PC provides data to or receives data from external devices or remote I/O Units, or otherwise handles data transactions for Link Systems.

set The process of turning a bit or signal ON.

set value The value from which a decrementing counter starts counting down or to which an incrementing counter counts up (i.e., the maximum count), or the time from which or for which a timer starts timing. Set value is abbreviated SV.

signal interrupt An interrupt caused by another task activating a `SIGNAL` instruction.

simple variable A non-array variable. Simple variables have only one value and cannot be subscripted.

single-precision constant Any number which is not specifically designated as an integer or double-precision floating point value, or which *is* designated as a single-precision value by a trailing exclamation point (`!`), or a numeric expression containing only integer and single-precision constants.

single-precision variable A variable that can hold a single-precision floating point value.

software error An error that originates in a software program.

software protect A means of protecting data from being changed that uses software as opposed to a physical switch or other hardware setting.

software switch See *memory switch*.

source code The code in which a program is written, e.g., ASCII. Source code must be converted to object code before execution.

Special I/O Unit A Unit that is designed for a specific purpose. Special I/O Units include Position Control Units, High-speed Counter Units, Analog I/O Units, etc.

SRAM Static random access memory; a data storage media.

SRQ See *service request*.

stack A data structure in memory which is maintained automatically by the BASIC Unit's CPU. The stack is used in `GOSUB` and `RETURN` instructions, as well as during interrupts.

statement The smallest complete unit of a BASIC program.

suboperand See *operand*.

subroutine A group of instructions placed separate from the main program and executed only when called from the main program or activated by an interrupt.

subscript An integer expression that designates an element of an array variable.

substitution statement A statement that uses the "=" operator to substitute the value of a second variable for that of the first variable.

SV Abbreviation for set value.

synchronous execution Execution of programs and servicing operations in which program execution and servicing are synchronized so that all servicing operations are executed each time the programs are executed.

syntax The form of a program statement (as opposed to its meaning). For example, the two statements, `LET A=B+B` and `LET A=B*2` use different syntaxes, but have the same meaning.

syntax error An error in the way in which a program is written. Syntax errors can include 'spelling' mistakes (i.e., a function code that does not exist), mistakes in specifying operands within acceptable parameters (e.g., specifying read-only bits as a destination), and mistakes in actual application of instructions (e.g., a call to a subroutine that does not exist).

system configuration The arrangement in which Units in a System are connected. This term refers to the conceptual arrangement and wiring together of all the devices needed to comprise the System. In OMRON terminology, system configuration is used to describe the arrangement and connection of the Units comprising a Control System that includes one or more PCs.

system error An error generated by the system, as opposed to one resulting from execution of an instruction designed to generate an error.

system error message An error message generated by the system, as opposed to one resulting from execution of an instruction designed to generate a message.

system variable A variable that contains information about the system (e.g. the current date and time, or the line number on which the last error occurred).

talker A device on a general-purpose interface bus that is sending data to other devices on the bus.

talker address The addresses on a general-purpose interface bus of a device that is sending data to other devices on the bus.

task A complete sub-unit within a BASIC program. Each task has its own variables, stack, and so on, and is completely independent of any other tasks in the program, although it may use inter-task communication to exchange data with these other tasks. The BASIC Unit can execute several tasks simultaneously.

task block Each task is delimited the `TASK` and `END TASK` statements; all statements between these statements are part of the task block.

task program A program written to perform a task.

terminator The code comprising an asterisk and a carriage return (* CR) which indicates the end of a block of data in communications between devices. Frames within a multi-frame block are separated by delimiters. Also a Unit in a Link System designated as the last Unit on the communications line.

three-line handshaking A handshaking method that uses three communications lines to perform handshaking.

timer A location in memory accessed through a TC bit and used to time down from the timer's set value. Timers are turned ON and reset according to their execution conditions.

timer interrupt An interrupt caused by the BASIC Unit's timer.

TR Area A data area used to store execution conditions so that they can be reloaded later for use with other instructions.

TR bit A bit in the TR Area.

transfer The process of moving data from one location to another within the PC, or between the PC and external devices. When data is transferred, generally a copy of the data is sent to the destination, i.e., the content of the source of the transfer is not changed.

transmission distance The distance that a signal can be transmitted.

UM area The memory area used to hold the active program, i.e., the program that is being currently executed.

uni-line message A message transferred on the control bus using only one signal line.

Unit In OMRON PC terminology, the word Unit is capitalized to indicate any product sold for a PC System. Though most of the names of these products end with the word Unit, not all do, e.g., a Remote Terminal is referred to in a collective sense as a Unit. Context generally makes any limitations of this word clear.

unit address A number used to control network communications. Unit addresses are computed for Units in various ways, e.g., 10 hex is added to the unit number to determine the unit address for a CPU Bus Unit.

unit number A number assigned to some Link Units, Special I/O Units, and CPU Bus Units to facilitate identification when assigning words or other operating parameters.

universal command A command sent to all devices on a general-purpose interface bus.

uploading The process of transferring a program or data from a lower-level or slave computer to a higher-level or host computer. If a Programming Devices is involved, the Programming Device is considered the host computer.

user indicator Indicators on a device that can be controlled by a user, e.g., from a user program being run on the device.

user program A program written by the user as opposed to programs provided with a product.

variable An area of memory in which a value can be stored; also refers to the name used in the program to designate that memory area.

variable-length character string A character string variable which can hold a string of any length (up to a system-defined maximum length).

volatile variable A variable which is not stored in battery-backed memory. Volatile variables lose their contents whenever power to the Unit is turned off.

watchdog timer A timer within the system that ensures that the scan time stays within specified limits. When limits are reached, either warnings are given or PC operation is stopped depending on the particular limit that is reached.

WDT See *watchdog timer*.

wildcard A special character used in a filename or extension to indicate zero or more possible characters.

wire communications A communications method in which signals are sent over wire cable. Although noise resistance and transmission distance can sometimes be a problem with wire communications, they are still the cheapest and the most common, and perfectly adequate for many applications.

word A unit of data storage in memory that consists of 16 bits. All data areas consists of words. Some data areas can be accessed only by words; others, by either words or bits.

word address The location in memory where a word of data is stored. A word address must specify (sometimes by default) the data area and the number of the word that is being addressed.

word allocation The process of assigning I/O words and bits in memory to I/O Units and terminals in a PC System to create an I/O Table.

work area A part of memory containing work words/bits.

work bit A bit in a work word.

work word A word that can be used for data calculation or other manipulation in programming, i.e., a ‘work space’ in memory. A large portion of the IR area is always reserved for work words. Parts of other areas not required for special purposes may also be used as work words.

write protect switch A switch used to write-protect the contents of a storage device, e.g., a floppy disk. If the hole on the upper left of a floppy disk is open, the information on this floppy disk cannot be altered.

write-protect A state in which the contents of a storage device can be read but cannot be altered.

Index

Revision History

A manual revision code appears as a suffix to the catalog number on the front cover of the manual.

Cat. No. W207-E1-2A

Revision code

The following table outlines the changes made to the manual during each revision. Page numbers refer to the previous version.

Revision code	Date	Revised content
1	June 1992	Original production
1A	November 1992	**Page 29:** Added paragraph to *Comments* of @ command. **Page 30:** Corrected notes under *Comments* for ALARM ON/OFF/STOP. **Page 36:** Corrected last 2 paragraphs of *Comments* for COM ON/OFF/STOP. **Page 50:** Corrected last 2 paragraphs of *Comments* for FINS ON/OFF/STOP. **Page 55:** Added note to *Comments* of IF THEN ELSE. **Page 81:** Added information to description of XON/XOFF and corrected first sentence of description of CS in table describing OPEN port specifications. **Page 86:** Corrected last 2 paragraphs of *Comments* for PC ON/OFF/STOP. **Page 91:** Removed "WAIT 10," from the fourth example. **Page 130:** Corrected last 2 paragraphs of *Comments* for ON SRQ GOSUB. **Pages 132 and 133:** Changed first paragraph in *Comments* and removed description of IEEE(8).
2	January 1993	*Appendix C Extended ASCII* has been added to the manual. **Page 7:** Allowable range for single-precision constants has been changed. The accuracy figures for double-precision constants has been changed. "disabled" changed to "stopped" throughout Section 3. **Pages 29, 99:** Changed *Purpose* and *Comments* for @ and PRINT/LPRINT. **Pages 35, 36, 67, 75, 118, 133:** Altered the program sample for CLOSE, LOG, ON FINS GOSUB, TAB, and STATUS. **Pages 36 to 38, 50, 61, 65, 68, 72, 74, 76 to 79, 85, 86, 116, 128, 129, 130, 133:** Changed *Comments* for COM ON/OFF/STOP, CONT, CSNG, FINS ON/OFF/STOP, KEY ON/OFF/STOP, LOF, MESSAGE, ON ALARM GOSUB, ON FINS GOSUB, ON KEY GOSUB, ON SIGNAL GOSUB, ON TIME$ GOSUB, ON TIMER GOSUB, PARACT, PC ON/OFF/STOP, POLL, STRING$, IRESET REN, ISET SRQ, and IEEE. **Pages 42, 45, 51, 60, 64, 86, 87, 117, 120, 124, 125, 127 to 131:** Added to *Comments* for DIM/RDIM, END PARACT, FOR TO STEP/NEXT, INTRB/INTRL/INTRR, LIST/LLIST, LOAD, PC READ, TAB, TRON, VERIFY, WHILE/WEND, CMD DELIM, CMD TIME-OUT, INPUT @, ISET IFC, LINE INPUT@, and PPOLL. **Pages 58, 93:** Added note to the end of INPUT $, and PC WRITE, **Page 77:** Note 3 changed for ON PC GOSUB. **Pages 80, 81, 114, 127, 130, 132, 133:** *Comments* for OPEN, STEP, CMD PPR, ON SRQ GOSUB, SRQ ON/OFF/STOP, and STATUS have been rewritten. **Page 110:** Added note and added information to the end of *Comments* for SAVE. **Page 135:** Diagram altered in *Integer Variable* of *Appendix A Memory Storage Format of Variables*.
2A	March 1993	Minor changes to add CV2000 and CVM1.